UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matej Mežik

Model za integracijo spletnih uporabniških vmesnikov na strani strežnika

MAGISTRSKO DELO

ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: prof. dr. Matjaž Branko Jurič

Ljubljana, 2015

IZJAVA O AVTORSTVU MAGISTRSKEGA DELA

Spodaj podpisani Matej Mežik, z vpisno številko **63060216**, sem avtor magistrskega dela z naslovom:

Model za integracijo spletnih uporabniških vmesnikov na strani strežnika.

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča;
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela;
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, 25. aprila 2015 Podpis avtorja:

Zahvaljujem se mentorju prof. dr. Matjažu Branku Juriču za strokovno vodenje in usmerjanje pri izdelavi magistrskega dela. Zahvaljujem se tudi svojim staršem, dekletu in prijateljem za moralno ter gmotno podporo med študijem.

Vsem skupaj iskrena hvala!

Kazalo

Slike

Izseki kode

Tabele

Seznam uporabljenih kratic

kratica	angleško	slovensko
ADF	Application Development Framework	Aplikacijsko razvojno okolje
AJAX	Asynchronous JavaScript and XML	Asinhroni klici z JavaScript in XML
API	Application Programming Interface	Aplikacijski programski vmesnik
ASPX	Active Server Page Framework	Ogrodje aktivnih strežniških strani
BPMN	Business Process Model and Notation	Model in zapis poslovnih procesov
CLI	Command Line Interface	Uporabniški vmesnik z ukazno vrstico
CRM	Customer Resource Management	Upravljanje s strankami
CSS	Cascading Style Sheets	Jezik za kaskadno oblikovanje
DiaMODL	Dialog Model Language	Jezik za opis modela dialoga
DOJO	Dojo Toolkit	Orodjarna Dojo
EAI	Enterprise Application Integration	Integracija aplikacij
ESB	Enterprise Service Bus	Storitveno vodilo
GUI	Graphical User Interface	Grafični uporabniški vmesnik
GWT	Google Web Toolkit	Spletna zbirka orodij podjetja Google
HRM	Human Resource Management	Upravljanje s človeškimi viri
HTML	Hypertext Markup Language	Označevalni jezik za oblikovanje večpredstavnostnih dokumentov
HTTP	Hypertext Transfer Protocol	Protokol za prenos hiperbesedila
HTTPS	Hypertext Transfer Protocol Secure	Protokol za varen prenos hiperbesedila
IDE	Integrated Development Environment	Integrirano razvojno okolje
IFML	Interaction Flow Modeling Language	Jezik za modeliranje interakcijskega toka
JAX-WS	Java API for XML Web Services	Javanski vmesnik za spletne storitve z uporabo strukture XML
JSF	Java Server Faces	Tehnologija za razvoj spletne aplikacije JSF
JSON	JavaScript Object Notation	Notacija za zapis objektov v uporabniku prijaznem formatu
JSP	Java Server Pages	Java strežniške strani
MARIA	Model-based Language for Interactive Applications	Jezik za opis modela interaktivne aplikacije
MDE	Model Driven Engineering	Modelno usmerjen razvoj
MVC	Model - View - Controller	Model - Pogled - Krmilnik

OOUI	Object-oriented User Interface	Objektno usmerjen uporabniški vmesnik
OWL	Web Ontology Language	Jezik spletnih ontologij
PLM	Product Lifecycle Management	Upravljanje življenjskega cikla produkta
POM	Project Object Model	Model za opis projekta
RDF	Resource Description Framework	Ogrodje za opis virov
REST	Representational State Transfer	Predstavitveni prenos stanja
RPC	Remote Procedure Call	Klic oddaljene procedure
RSS	Rich Site Summary	Protokol za objavo in distribucijo spletnih vsebin v zapisu XML
SaaS	Software as a Service	Programska oprema kot storitev
SCM	Supply Chain Management	Upravljanje dobavne verige
SOA	Service Oriented Architecture	Storitveno usmerjena arhitektura
SOAP	Simple Object Access Protocol	Protokol za izmenjavo vsebine XML s spletno storitvijo
SPARQL	SPARQL Protocol and RDF Query Language	Jezik za izvajanje poizvedb nad semantično opisanimi podatki
SQL	Structured Query Language	Strukturiran poizvedovalni jezik
TCP/IP	Transmission Control Protocol/Internet Protocol	Povezani protokol transportnega sloja v skladu s TCP/IP
TUI	Touch User Interface	Uporabniški vmesnik, občutljiv za dotik
UDDI	Universal Description, Discovery and Integration	Univerzalni opis, raziskovanje in integracija
UML	Unified Modeling Language	Poenoten jezik za modeliranje
UsiXML	User Interface Extensible Markup Language	Razširjen označevalni jezik za uporabniške vmesnike
WebML	Web Modeling Language	Jezik za spletno modeliranje
WS-BPEL	Web Service Business Process Execution Language	Programski jezik, namenjen izvajanju poslovnih procesov, kot spletna storitev
WSDL	Web Service Description Language	Jezik za opis spletnih storitev
WSRP	Web Services for Remote Portlets	Spletne storitve za oddaljene portale
WUIIDL	Web User Interface Integration Description Language	Jezik za opis integracije spletnih uporabniških vmesnikov

XAML	Extensible Application Markup Language	Razširljivi jezik, namenjen opisu aplikacij
XHTML	Extensible Hypertext Markup Language	Razširjen označevalni jezik za oblikovanje večpredstavnostnih dokumentov
XIML	Extensible Interface Markup Language	Razširljivi jezik, namenjen opisu vmesnikov
XJC	JAX Binding Compiler	Prevajalnik za izvajanje vezave
XML	Extensible Markup Language	Razširljiv označevalni jezik
XPIL	Extensible Presentation Integration Language	Razširjen jezik za integracijo na predstavitvenem nivoju
XSD	XML Schema Definition	Definicija sheme XML

Povzetek

Uporabniški vmesnik predstavlja glavno stično točko med uporabnikom in aplikacijo. Z namenom poenostavitve njegove uporabe in izboljšanja uporabniške izkušnje nastajajo pristopi ter modeli za učinkovitejši razvoj uporabniških vmesnikov. Tako je nastal tudi pristop z integracijo aplikacij na različnih nivojih. V magistrskem delu smo se osredotočili izključno na integracijo spletnih uporabniških vmesnikov. Predstavili smo lasten model integracije spletnih uporabniških vmesnikov, ki uporablja opisni jezik WUIIDL (*Web User Interface Integration Description Language*). Jezik je bil razvit v okviru magistrskega dela, z njim smo opisali ključne točke integracije. Strukturo jezika smo definirali s pomočjo sheme XSD (*XML Schema Definition*), ki posredno predstavlja strukturo vhodnih podatkov. Komunikacijo med posameznimi komponentami modela smo izvedli z uporabo spletnih storitev, ki smo jih opisali z datoteko WSDL (*Web Service Description Language*). V sklopu magistrskega dela smo pripravili prototip, njegova implementacija pa temelji na platformi Java EE. Integracijo smo izvedli z uporabo ogrodja Apache Wicket, ki nam je bilo v pomoč pri prikazovanju komponent na spletnem uporabniškem vmesniku. V prototipu smo podprli tudi postopek validacije, in sicer na podlagi vhodnih podatkov, podanih z opisnim jezikom WUIIDL. Model smo ocenili na podlagi testiranja množice primerov aplikacij SaaS (*Software as a Service*), s katerimi smo pripravili nabor štirih scenarijev, ki so pogosto v uporabi. Ugotovili smo, da model naslavlja ključne dele, potrebne za zagotavljanje celovite integracije.

Ključne besede: integracija, spletni uporabniški vmesnik, definiranje opisnega jezika, prikazovanje komponent, validacija.

Abstract

User interface is the main point of contact between the user and the application. With an aim to simplify its use and improve the user experience different approaches and models have been proposed for a more efficient development of user interfaces. This has resulted in approaches for application integration at different levels. In this Master's thesis, we are focusing exclusively on the integration of web user interfaces. We present our own model for web user interface integration, which uses WUIIDL (*Web User Interface Integration Description Language*). The WUIIDL language has been developed within the context of the Master's thesis to describe the integration key points. The structure of the language was defined using XSD (*XML Schema Definition*) schemas and indirectly represents the structure of the input data. Communication between the individual components of the model was performed using web services which are described with WSDL (*Web Service Description Language*) file. We have prepared a prototype, its implementation being based on the Java EE platform. Integration was performed using Apache Wicket framework which has been helpful in displaying the components on the web user interface. The prototype also supports the validation process, namely based on the input data provided using the WUIIDL description language. The model was evaluated by testing a set of SaaS (*Software as a Service*) application examples used to prepare a set of four commonly used scenarios. We found that the model addresses the key parts necessary to ensure comprehensive integration.

Keywords: integration, web user interface, description language definition, display of components, validation.

1 Uvod

Uporabniški vmesniki so glavna stična točka med uporabnikom in aplikacijo, kjer je njihova vloga zagotavljati interakcijo uporabnika z aplikacijo. Na takšen način uporabnik upravlja aplikacijo, ki mu prek uporabniškega vmesnika posreduje informacije, kot rezultat uporabnikove akcije. Poznamo različne tipe uporabniških vmesnikov, ki jih ločimo glede na njihove lastnosti, kot so način prikaza informacij, način interakcije, namen uporabe, in na kateri tehnologiji temeljijo.

V magistrskem delu smo se ukvarjali s problematiko integracije spletnih uporabniških vmesnikov. Naslovili smo ključne izzive pri postopku integracije uporabniških vmesnikov, ki smo jih v magistrskem delu razčlenili in pripravili model, s katerim smo podprli integracijo uporabniških vmesnikov. Glavna komponenta modela je opisni jezik WUIIDL (*Web User Interface Integration Description Language*), s katerim je možno opisati ključne točke integracije, torej uporabniški vmesnik in njegove lastnosti. Osredotočili smo se izključno na spletne uporabniške vmesnike, saj trenutno večina aplikacij temelji na konceptu odjemalec–strežnik (*client–server*), predvsem zaradi možnosti oddaljenega dostopa in zaradi naraščanja števila prenosnih naprav.

Postopek integracije se sicer uporablja na več področjih, tako v ekonomiji kot tudi v upravljanju podjetij in informatiki. Integracija v osnovi namreč pomeni združitev sorodnih delov, ki so povezani prek skupnih točk in tako zagotavljajo celostno koordinacijo [29]. V primeru integracije aplikacij moramo omeniti, da obstaja več nivojev tovrstne integracije, in sicer na podatkovnem nivoju, ki je namenjena skupnemu upravljanju s podatki, na poslovnem nivoju, kjer si aplikacije delijo skupno poslovno logiko, in pa na nivoju uporabniških vmesnikov. V zadnjem času se pogosto uporablja tudi integracija na nivoju procesov. V magistrskem delu smo se ukvarjali izključno z integracijo na nivoju uporabniških vmesnikov.

Če želimo opraviti pregled stanja na področju integracije spletnih uporabniških vmesnikov, moramo najprej pojasniti nekaj primerov uporabe, prek katerih lahko ugotovimo težavnost tovrstne integracije. V podjetjih pogosto opazimo, da so aplikacije ločene glede na njihov namen oz. uporabo [15]. Tako najdemo aplikacije za pomoč pri upravljanju poslovanja podjetja, ki je sestavljeno iz različnih modulov, kot so modul za upravljanje s strankami CRM (*Customer Resource Management*), modul za upravljanje s človeškimi viri HRM (*Human Resource Management*), modul za upravljanje dobavne verige SCM (*Supply Chain Management*) in modul za upravljanje življenjskega cikla produkta PLM (*Product Lifecycle Management*). Pri tem se dogaja, da mora uporabnik ali več njih pogosto vnesti enak podatek v več različnih aplikacij. To težavo bi sicer lahko odpravili že z integracijo na podatkovnem nivoju, vendar je ta velikokrat neizvedljiva, saj morajo biti vse aplikacije seznanjene z njihovimi podatkovnimi modeli. Slednje pomeni, da bi morale vse aplikacije imeti skupen podatkovni model, ki bi ga

sicer lahko uvedli z uporabo transformacije. Razvoj aplikacij bi moral tako temeljiti na enem samem podatkovnem modelu, kar pa je v praksi težko izvedljivo, saj je razvoj aplikacij največkrat porazdeljen med več podjetij ali pa je časovno zamaknjen, pri tem pa vzdrževanje podatkovnega modela postane velik izziv.

Glede na to, da so uporabniški vmesniki običajno dinamični in sestavljeni iz več komponent oz. delov [6], smo se morali najprej osredotočiti na problem, kako opisati uporabniški vmesnik, vključno s pripadajočimi dogodki, da bo opis čim bolj natančen in še vedno razumljiv. Glavni cilj magistrskega dela je torej definirati strukturo opisnega jezika, s katerim lahko opišemo spletne uporabniške vmesnike, ki jih želimo vključiti v model za integracijo spletnih uporabniških vmesnikov. Opisni jezik predstavlja temelj modela, za katerega smo pripravili arhitekturo in prototip, z namenom ugotavljanja njegove ustreznosti.

Razvoj opisnega jezika predstavlja glavni prispevek magistrskega dela. S pomočjo opisnega jezika lahko namreč posredujemo vhodne podatke, na podlagi katerih se kasneje izvede proces integracije. Ti podatki vsebujejo informacije o komponentah, njihovih postopkih validacije in splošne informacije o spletnem uporabniškem vmesniku. Naslednji prispevek je arhitektura modela za izvedbo integracije, ki je v osnovi razdeljena na zbiranje vhodnih podatkov, njihovo transformacijo v prikaz spletnega uporabniškega vmesnika in na koncu tudi sprejem podatkov, vključno s fazo validacije. Z validacijo smo se ukvarjali ob koncu magistrskega dela, definirali smo postopke za preverjanje skladnosti podatkov, saj je njihova vrednost ključnega pomena za nadaljnje korake distribucije podatkov na ciljne lokacije.

Pred razvojem modela smo pregledali področje in identificirali ključne izzive, ki so za nas predstavljali cilje magistrskega dela. Tako smo v poglavju 2 pregledali tipe uporabniških vmesnikov in njihove lastnosti. Pregledali smo tudi različne vrste integracije, predvsem s stališča integracije aplikacij. Ugotavljali smo glavne pomanjkljivosti obstoječih rešitev in pri tem našli več skupnih točk z našim modelom. Nato smo definirali model za integracijo spletnih uporabniških vmesnikov, in sicer v poglavju 3, kjer smo opisali posamezne dele modela. Po uspešni definiciji modela smo v poglavju 4 predstavili praktični del magistrskega dela, v katerem smo pripravili prototip modela in z njim ugotovili njegovo ustreznost. V poglavju 5 smo posebno pozornost namenili evalvaciji modela, ki smo ga razvili in zanj pripravili prototip. Za lažjo predstavitev smo izbrali ustrezen nabor scenarijev, s pomočjo katerih smo lažje ocenili model in s tem tudi njegovo ustreznost oz. učinkovitost. Identificirali smo tudi možnosti izboljšav in jih primerno opisali. Na koncu smo v poglavju 6 predstavili svoje poglede in ugotovitve s stališča modela ter njegovega prototipa.

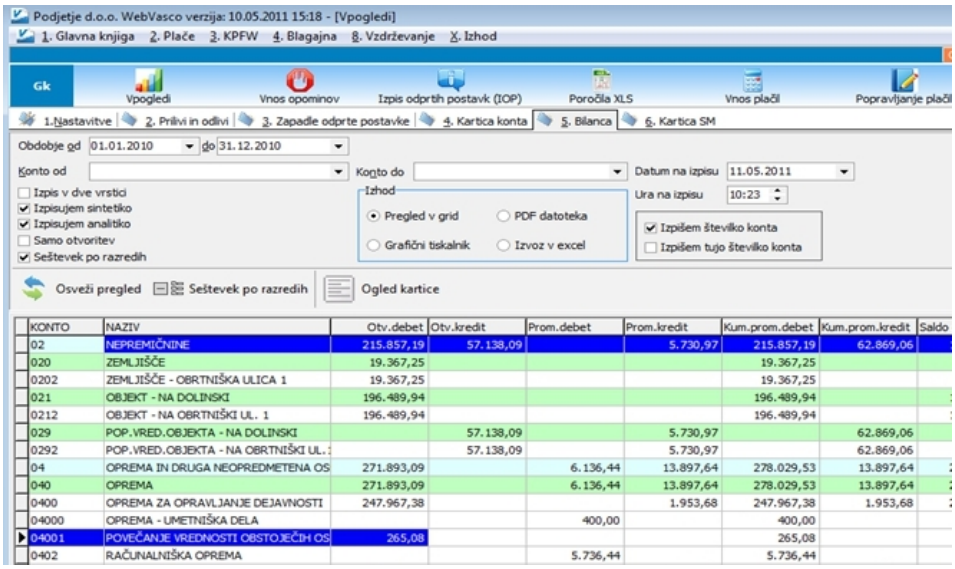
2 Pregled uporabniških vmesnikov

Uporabniški vmesnik predstavlja stično točko med uporabnikom in aplikacijo, na kateri teče interakcija uporabnika z aplikacijo. Interakcija lahko poteka prek tekstovnih ukazov [30], kar omogoča ukazna vrstica, ali pa s pomočjo komponent, ki sestavljajo grafični vmesnik. Uporabniške vmesnike lahko torej delimo glede na tip interakcije [8], glede na način prikaza informacij, glede na namen uporabe ali pa glede na tehnologijo [32]. Osredotočili se bomo na grafične uporabniške vmesnike, med katere sodijo tudi spletni uporabniški vmesniki.

2.1 Tipi uporabniških vmesnikov

2.1.1 Pregled tipov uporabniških vmesnikov

Poznamo več tipov uporabniških vmesnikov. V osnovi jih delimo na grafične uporabniške vmesnike (*Graphical User Interfaces - GUI*) (Slika 2.1) in vmesnike z ukazno vrstico (*Command Line Interfaces - CLI*) [32], poznamo pa tudi druge vmesnike, ki s stališča informatike nimajo posebnega pomena. Grafične vmesnike lahko delimo tudi glede na namen in način uporabe.



Slika 2.1: Grafični uporabniški vmesnik (*vir: www.vasco.si*).

Tako najdemo uporabniške vmesnike, ki so namenjeni izključno neposrednemu upravljanju objektov (*Direct Manipulation Interface*) [7], katerih odziv uporabnik lahko opazi praktično v realnem času. Takšne uporabniške vmesnike, ki jim strokovno pravimo tudi objektno usmerjeni uporabniški vmesniki (*Object-Oriented User Interfaces - OOUI*) [32], uporabljajo predvsem orodja za oblikovanje (*Adobe Photoshop*, *Maya*) in v primeru igralnih uporabniških vmesnikov, kjer si uporabnik lahko nastavi videz objekta, ki ga želi med igranjem upravljati. Klasični uporabniški vmesniki so implementirani za uporabo kazalca miške, novejši vmesniki pa so prilagojeni tudi za upravljanje z dotikom (*Touch User Interface - TUI*) [20]. Takšni uporabniški

vmesniki torej za upravljanje potrebujejo specifično prikazovalno napravo (*touchscreen*), ki je občutljiva za dotike. V industriji so v uporabi uporabniški vmesniki, ki niso namenjeni obojestranski interakciji, temveč le enostranski, in sicer imajo nalogo prikazovanja opozoril (*Attentive User Interfaces*) [13], ki jih mora uporabnik upoštevati pri nadaljnjem delu. Pri paketni obdelavi podatkov aplikacija od uporabnika pričakuje začetne parametre, nato pa na uporabniškem vmesniku prikazuje le še status obdelave. V ta namen obstajajo posebni uporabniški vmesniki za paketno obdelavo (*Batch Interfaces*), ki od uporabnika pričakujejo interakcijo samo za nastavljanje parametrov in zagon obdelave.

V zadnjem času so z namenom približevanja uporabniku nastali posebni uporabniški vmesniki, ki želijo od uporabnika izzvati njegovo pozornost. To dosežejo s pomočjo poosebljanja uporabniškega vmesnika (*Conversational Interface Agents*), in sicer z animacijo virtualnega bitja, s katerim uporabnik komunicira s pomočjo pogovora. Ta poteka v tekstovni obliki, pri tem pa imajo pomembno vlogo animacije, ki predstavljajo odziv na uporabnikovo dejanje. V Sloveniji smo pred leti na spletni strani Finančne uprave RS (FURS) dobili primer uporabniškega vmesnika, na katerem je virtualna davčna asistentka Vida (Slika 2.2) odgovarjala na vprašanja davčnih zavezancev.

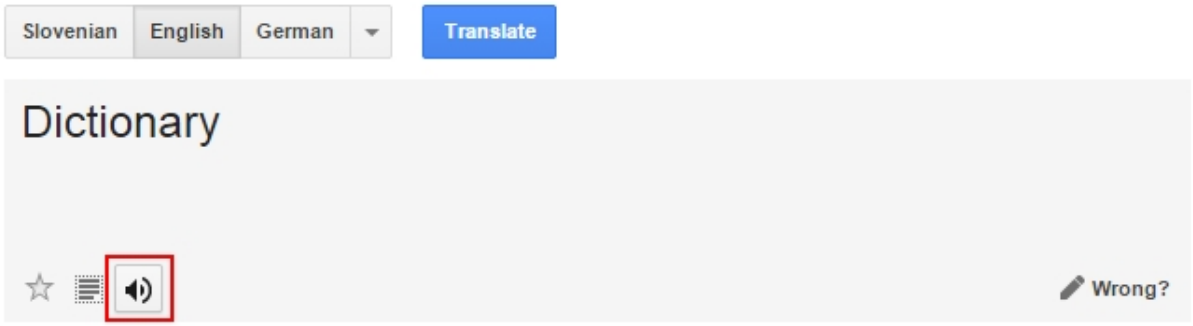
Slika 2.2: Virtualna davčna svetovalka VIDA (*vir: www.fu.gov.si*).

Za zabavo in spodbujanje gibalnih veščin med sproščanjem so nastali uporabniški vmesniki, katerih interakcija uporabnika temelji na njegovih gibih (*Motion Tracking Interfaces*). Z gibanjem simulira gibanje objekta na uporabniškem vmesniku. Omenjeni način interakcije je v osnovi nadgradnja uporabniških vmesnikov, ki so občutljivi za poteze (*Gesture Interfaces*). Uporabnik lahko naredi potezo z miško ali z naprednejšo napravo, kot je plošča za risanje (*Drawing Tablet*).

Načini interakcije uporabnika z aplikacijo so postajali vedno bolj dovršeni, zato je nastal poseben tip uporabniškega vmesnika, ki za interakcijo uporabnika uporablja senzorje (*Zero-Input interfaces*), s katerimi identificira uporabnikove akcije. Nadgradnja omenjenega pristopa predvideva interakcijo brez eksplicitnih ukazov (*Non-command User Interfaces*) uporabnika. Temelji na spremljanju njegovih potreb in namenov, na podlagi katerih nato ugotovi, katera bo uporabnikova naslednja akcija.

Povsem drugačen namen imajo refleksivni uporabniški vmesniki (*Reflexive User Interfaces*), katerih posebnost je možnost nastavljanja pomena ukazov, torej uporabnik lahko sam določi, kaj bo pomenil določeni ukaz. Takšni uporabniški vmesniki so sestavni del razvojnih okolij, kjer si vsak razvijalec lahko nastavi poljubno shemo ukazov. V teh okoljih se pojavljajo tudi uporabniški vmesniki, ki temeljijo na nalogah (*Task-Focused Interfaces*), informacije pa se obdelujejo v obliki nalog, ki predstavljajo primarno enoto interakcije.

Če zaidemo tudi na področje medijev, lahko naletimo na uporabniške vmesnike, ki so namenjeni izključno pregledovanju podrobnosti opazovanega objekta (*Zooming User Interfaces*). Pri tem lahko uporabnik sam izbere stopnjo podrobnosti, ki ga zanimajo. Posebnost imajo tudi vmesniki, ki temeljijo na glasovnem odzivu (*Voice User Interfaces*), interakcija z glasovnim sporočanjem je lahko dvosmerna, od uporabnika k vmesniku ali obratno, lahko pa je samo enosmerna. Primer omenjenega uporabniškega vmesnika lahko najdemo na spletni strani podjetja Google, ki poleg prevajanja nudi tudi možnost glasovne izgovarjave tujih besed (Slika 2.3).



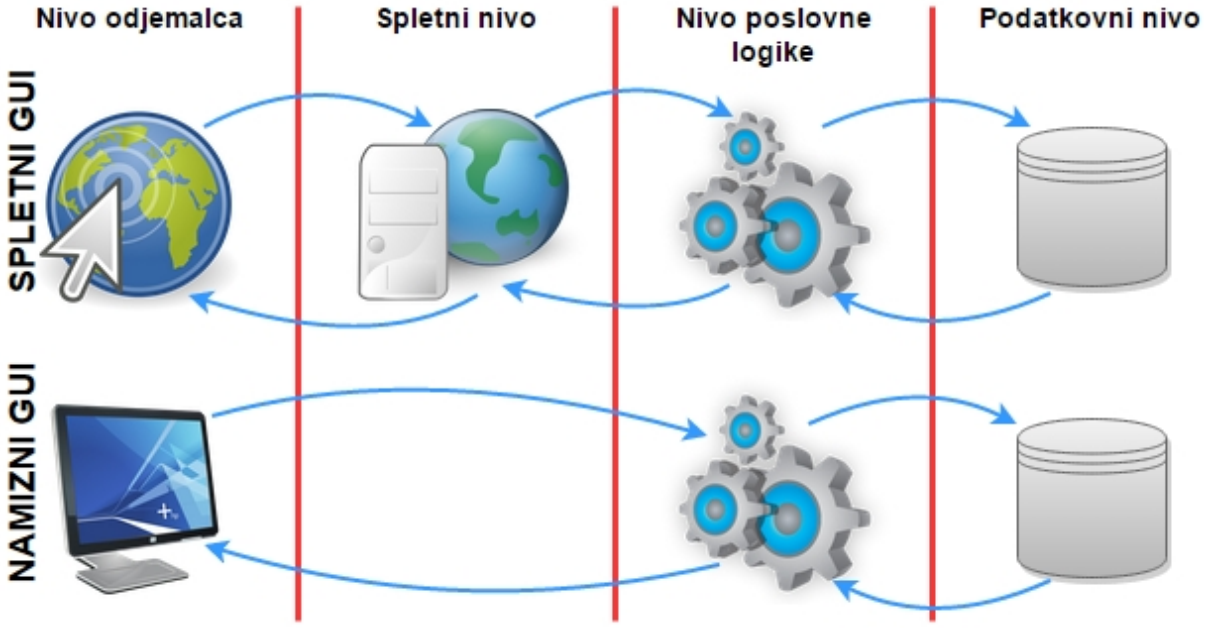
Slika 2.3: Uporabniški vmesnik z glasovno izgovorjavo (*vir: www.google.com*).

Pri opisu grafičnih uporabniških vmesnikov ne smemo pozabiti na njihovo delitev glede na uporabljeno tehnologijo in arhitekturo. Ločimo namreč med namiznimi in spletnimi grafičnimi vmesniki. Posebnost spletnih grafičnih vmesnikov je v tem, da uporabnik običajno nima možnosti neposrednega vpliva na njihovo odzivnost, saj je poslovna logika aplikacije največkrat nameščena na oddaljenem strežniku, do katerega pa uporabnik nima neposrednega dostopa. Obstajajo tudi izjeme, kot sta skriptni jezik JavaScript in javanski programčki (*Java Applets*), pri katerih se poslovna logika izvaja na strani odjemalca.

2.1.2 Primerjava grafičnih uporabniških vmesnikov

Ugotovili smo že, da nas obdaja več tipov uporabniških vmesnikov, ki so prilagojeni za različne potrebe in različna okolja. Tako so grafični uporabniški vmesniki namenjeni prikazovanju informacij uporabniku na čim bolj uporaben način, medtem ko je ukazna vrstica namenjena uporabnikom, ki jim oblika uporabniškega vmesnika in posledično tudi oblika prikaza informacij nista tako pomembni. Uporabniški vmesniki se razlikujejo tudi glede na tehnologije, na katerih temelji njihova arhitektura.

Prikazovanje spletnih grafičnih vmesnikov zahteva drugačen pristop, v katerem ima pomembno vlogo predvsem odjemalec, s katerim uporabnik dostopa do spletne aplikacije. Spletni uporabniški vmesniki so namreč odvisni od spletnih tehnologij, ki odgovornost izvajanja delijo na dve strani (Slika 2.4), in sicer odgovornost na strani strežnika in odgovornost na strani odjemalca. Strežnik ima pri tem vlogo koordinatorja priprave ustreznih odzivov na zahtevke, ki jih pošilja uporabniški vmesnik.



Slika 2.4: Primerjava konceptov spletnih GUI z namiznimi GUI.

Odgovornost priprave spletnega uporabniškega vmesnika je tako lahko na strani strežnika ali na strani odjemalca. Temu primerno so prilagojene tudi spletne tehnologije, ki jih tako delimo na tiste, ki pripravijo vsebino spletnega uporabniškega vmesnika na strani strežnika (*Server-side Web Development*), in tiste, ki jo pripravijo na strani odjemalca (*Client-side Web Development*).

V primeru priprave vmesnika na strani strežnika razvijalec uporablja tehniko razvoja spletnih uporabniških vmesnikov, ki poskrbi, da se rezultat izvajanja akcije odraža kot prilagojen spletni uporabniški vmesnik. Njegov videz je odvisen od množice vhodnih parametrov, med katere sodi tudi status, v katerem je trenutno uporabnik. V ta namen so bila

razvita različna ogrodja ali tehnologije, ki za izvajanje uporabljajo strežniško gostovanje, med bolj znanimi pa so JSP (*Java Server Pages*), JSF (*Java Server Faces*), Java Servlets, ASPX (*Active Server Page Framework*) in Apache Wicket. Z uporabo takšnih ogrodij je enostavneje razviti spletno aplikacijo skupaj z uporabniškim vmesnikom. Takšno ogrodje smo za potrebe razvoja prototipa uporabili tudi sami, in sicer ogrodje Apache Wicket. Podrobneje ga bomo spoznali v opisu praktične rešitve (poglavje 4), sicer pa temelji na programskem jeziku Java in predlogah za lažje strukturiranje uporabniškega vmesnika. Ob tem poskrbi tudi za komunikacijo med uporabniškim vmesnikom in strežnikom glede na izbrane akcija uporabnika.

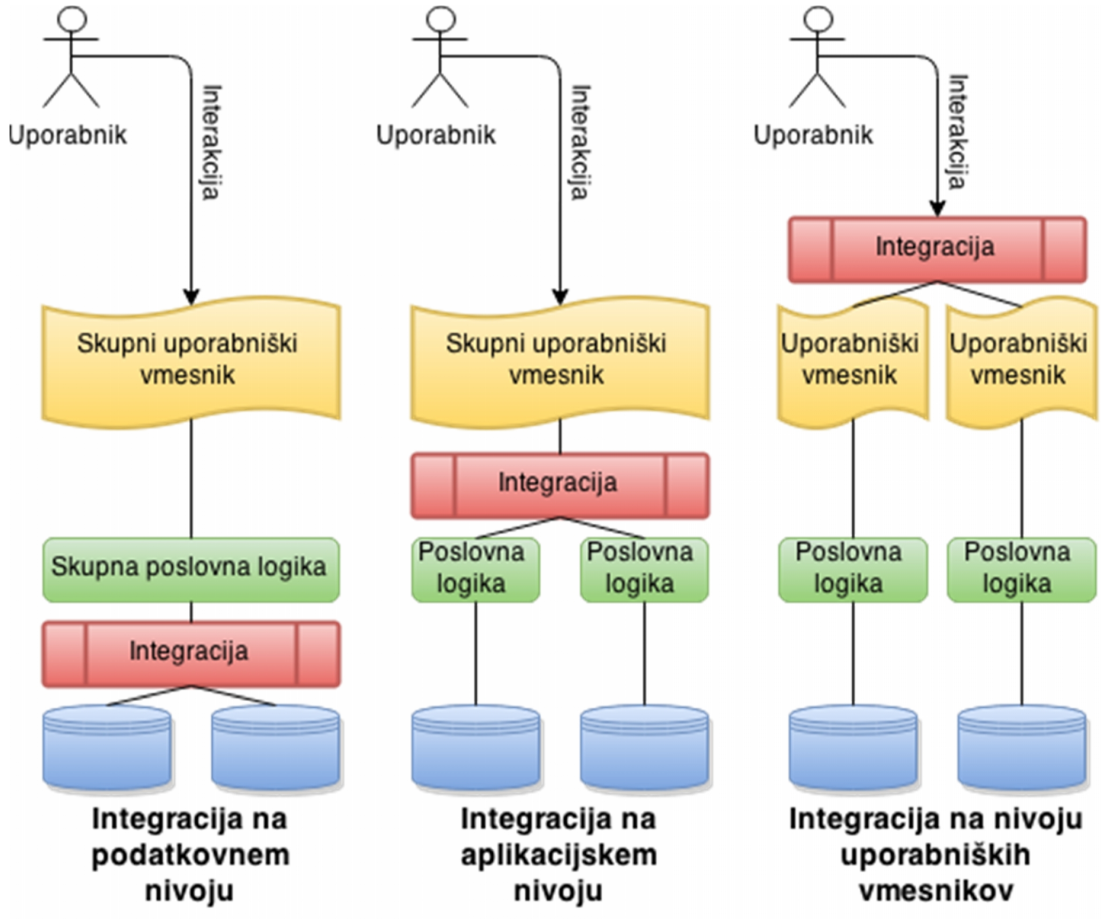
O izvajanju uporabniškega vmesnika na strani brskalnika (ima vlogo odjemalca) pa govorimo takrat, ko gre predvsem za programsko kodo, ki jo brskalnik prejme v odgovoru strežnika in jo izvaja pod lastnim okriljem, rezultat česa je prikaz uporabniškega vmesnika. Torej se uporabniški vmesnik pripravi šele na strani odjemalca, ti pa podpirajo predvsem izvajanje skriptnih jezikov. Tehnologije, ki so namenjene takšnemu razvoju, so označevalni jezik HTML5 (*Hypertext Markup Language*) skupaj s skriptnim jezikom JavaScript in ogrodja Bootstrap, DOJO (*Dojo Toolkit*), jQuery, AngularJS, GWT (*Google Web Toolkit*) in Vaadin. Slednja pri tem izstopata, saj razvijalcu ponujata možnost razvoja v programskem jeziku Java, nato pa izvedeta neposredno pretvorbo v skriptni jezik JavaScript. Glavna prednost uporabe takšnih ogrodij je v tem, da ogrodje že prek svojih vmesnikov poskrbi za generiranje ustreznih odzivov na akcije uporabnika, sicer bi moral razvijalec za to poskrbeti ročno.

Za prikazovanje spletnega uporabniškega vmesnika je obvezna uporaba spletnega odjemalca, ki se uporablja za prikaz podatkov v do uporabnika prijaznejši obliki, torej v obliki uporabniškega vmesnika. Pri tem je treba poudariti, da za združljivost med posameznimi komponentami arhitekture skrbijo številni protokoli, ti pa definirajo koncepte, ki trenutno obvladujejo svetovni splet. Spletni uporabniški vmesniki, ki jih med drugim prav tako uvrščamo med grafične uporabniške vmesnike, so prilagojeni in hkrati tudi omejeni z množico protokolov ter pravil, ki določajo koncept in arhitekturo vmesnika.

Razlika med namiznimi in spletnimi uporabniškimi vmesniki je tudi v tem, da se komponente, ki sestavljajo spletni uporabniški vmesnik, ne zavedajo sosednjih komponent. Spletni uporabniški vmesnik je v osnovi namreč rezultat interpretacije jezika za označevanje HTML. Komunikacijo med komponentami sicer lahko vzpostavimo s pomočjo strežnika ali s pomočjo skriptnih jezikov (*JavaScript*, *jQuery*), ki se izvajajo na strani odjemalca. Grafični elementi spletnih uporabniških vmesnikov namreč delujejo v okviru brskalnika (odjemalca), medtem ko namizni uporabniški vmesniki tipično (razen ogrodja Java Swing) uporabljajo grafične elemente, ki jih zagotavlja operacijski sistem.

2.2 Integracija aplikacij

Klasična arhitektura aplikacij je razdeljena na več nivojev, pri čemer je uporabniški vmesnik najvišji nivo arhitekture, shramba podatkov pa najnižji nivo. Poznamo integracijo aplikacij na različnih nivojih (Slika 2.5) [24], tako lahko izvedemo integracijo na podatkovnem nivoju, in sicer s pomočjo povezovanja podatkovnih baz. Podobno je z integracijo na aplikacijskem nivoju, kjer uporabniški vmesnik komunicira z več aplikacijami prek aplikacijskih vmesnikov. Pri poslovnih informacijskih sistemih imajo glavno vlogo poslovni procesi, s pomočjo katerih so definirani različni postopki za doseganje določenega cilja. Tudi na področju poslovnih procesov se je pojavil koncept, ki opisuje njihovo integracijo [25], in sicer z uporabo medsebojne komunikacije. Integracija na nivoju procesov je navzoča predvsem v informacijskih rešitvah večjih korporacij, medtem ko integracija na nivoju uporabniškega vmesnika še vedno izkazuje določene pomanjkljivosti, ki so posebej izrazite pri uporabi spletnih uporabniških vmesnikov. V delu se bomo osredotočili izključno na izzive, povezane z integracijo spletnih uporabniških vmesnikov, saj trenutno prevladujejo v množici vseh uporabniških vmesnikov.



Slika 2.5: Nivoji integracije aplikacij.

2.2.1 Integracija aplikacij na podatkovnem nivoju

Ko aplikacija uporablja več različnih podatkovnih baz, govorimo o integraciji na podatkovnem nivoju. To pomeni, da se podatki hranijo porazdeljeno v več podatkovnih bazah hkrati. Pri tem je treba poskrbeti za ustrezno poenotenje podatkovnih modelov na relaciji med aplikacijskim in podatkovnim nivojem, kar je izvedljivo na več načinov. Prepisovanje podatkov v skupno podatkovno bazo [11] je sicer ena izmed rešitev, vendar vsebuje ovire, kot je zagotavljanje konsistentnosti podatkov. Pristop s poenotenim dostopom do podatkovnih baz pomeni, da aplikacija dostopa do podatkov le prek točke, ki predstavlja virtualen dostop do vseh podatkovnih baz. Prednost je v tem, da aplikacija dostopa do fizično sicer porazdeljenih podatkov prek skupne točke, še vedno pa se mora zavedati podatkovnih modelov vseh podatkovnih baz. Poleg tega je omenjena metoda časovno zelo potratna. Uporaba posebej v ta namen pripravljenih orodij, ki podatke pripravijo glede na vnaprej definirane podatkovne modele, predstavlja dodatno možnost za integracijo na podatkovnem nivoju. V tem primeru aplikacija dostopa do podatkov neposredno s pomočjo orodij, ki predstavljajo vmesnik za komuniciranje s podatkovnimi bazami. Vmesnik se mora tako zavedati vseh podatkovnih modelov, ki jih preslika v skupen podatkovni model, razumljiv tudi aplikaciji.

2.2.2 Integracija na nivoju poslovne logike

Nivo poslovne logike predstavlja jedro aplikacije, saj je v njem večina odločitvene logike. Običajno vsebuje glavni del programske kode, ki komunicira spodaj s podatkovnim nivojem, zgoraj pa z uporabniškim vmesnikom. Na nivoju poslovne logike je aplikacija razdeljena na komponente, in sicer glede na vsebino, ki jo posamezna komponenta pokriva. Omenjene komponente imajo definirane aplikacijske vmesnike, ki jim rečemo API (*Application Programming Interface*) [2]. Aplikacijski vmesnik določa podatkovni model in komunikacijski protokol, ki je dovoljen na izbranem vmesniku. Integracija na nivoju poslovne logike je torej izvedena tako, da aplikacija uporablja aplikacijske vmesnike druge aplikacije. Med pristope, ki temeljijo na integraciji z uporabo programskih vmesnikov, uvrščamo tudi model SOA (*Service Oriented Architecture*). Prednost integracije na nivoju poslovne logike [24] je v tem, da omogoča enostavno prilagajanje poslovne logike znotraj posamezne komponente, vmesnik pa običajno ostane nespremenjen. S tem razlogom integracija na nivoju poslovne logike predstavlja pomemben vzorec razvoja modernih aplikacij.

2.2.3 Integracija aplikacij na nivoju procesov

Modernejši pristopi razvoja informacijskih sistemov temeljijo na podpori uporabe kompozitnih aplikacij, v kombinaciji z obstoječimi storitvami. Za doseganje boljših poslovnih rezultatov so podjetja začela razmišljati v smeri komunikacije med poslovnimi procesi, in sicer na nivoju realnega časa. Pri tem se je pojavila težava, informacijski sistem podjetja je namreč največkrat združeval več različnih aplikacij, ki so delovale popolnoma neodvisno druga poleg

druge in so podpirale le določene poslovne procese. Tako je nastal koncept integracije na nivoju procesov, ki temelji na komunikaciji med procesi. Komunikacija poteka s pomočjo izmenjave sporočil, in sicer prek storitvenega vodila ESB (*Enterprise Service Bus*) [12], ki skrbi za obdelavo sporočil. Sporočila lahko predstavljajo zahtevek ali odgovor. V primeru zahtevka storitveno vodilo poskrbi za izvedbo klica določene storitve na podlagi parametrov, podanih v zahtevku, medtem ko sporočilo za odgovor pripravi glede na rezultat klica storitve. Uspešnost integracije na nivoju procesov je pogojena z delno integracijo na podatkovnem nivoju [3]. Z njeno pomočjo procesom zagotovimo razumevanje podatkovnega modela preostalih procesov, s katerimi želi komunicirati. Dokler se procesi ne zavedajo pomena podatkov, ki jih prejemajo od ostalih procesov, integracija na nivoju procesov nima pravega pomena. Z integracijo poslovnih procesov je moč doseči učinkovitejše zbiranje informacij, ki zaradi optimizacije izvajanja procesov hitreje dosežejo svoje cilje.

2.2.4 Integracija aplikacij na nivoju uporabniških vmesnikov

Integracija na nivoju uporabniških vmesnikov pomeni združevanje več uporabniških vmesnikov v skupnega. To pomeni, da je treba vse funkcionalnosti obstoječih uporabniških vmesnikov s pomočjo integracije podpreti tudi na skupnem uporabniškem vmesniku.

Za doseganje cilja integracije je treba najprej narediti analizo obstoječih uporabniških vmesnikov, s pomočjo katere identificiramo posamezne sestavne dele uporabniških vmesnikov. Njihovo sestavo lahko razdelimo na več sklopov, in sicer na komponente, ki predstavljajo glavni del uporabniških vmesnikov, dogodke, ki so posledica interakcije uporabnika z uporabniškim vmesnikom, in prikaz informacij, čemur je uporabniški vmesnik v osnovi tudi namenjen. Uporabniški vmesniki naj ne bi vsebovali poslovne logike, vendar pa se včasih temu ni moč izogniti, še posebno pri izvajanju validacije podatkov. To se pogosto pojavlja pri spletnih uporabniških vmesnikih, kjer se na takšen način izognemo redundanci poslanih zahtevkov.

Iz faze analize je moč opaziti, da za izvedbo integracije potrebujemo mehanizem, ki bo skrbel za opis sestavnih delov spletnih uporabniških vmesnikov. Pristop, kako bomo pridobili informacije iz obstoječih uporabniških vmesnikov, je lahko različen in hkrati odvisen od stopnje avtomatizacije. Če želimo razčlenjevanje uporabniških vmesnikov opravljati v realnem času, potrebujemo razčlenjevalnik (*parser*), ki bo preiskoval uporabniške vmesnike in zbrane informacije zapisoval v obliki, primerni za integracijo. Obliko je treba določiti z definicijo opisnega jezika. Informacije, podane z opisnim jezikom, predstavljajo vhodne podatke za integracijo uporabniških vmesnikov.

Postopek integracije spletnih uporabniških vmesnikov se torej začne z zbiranjem informacij o obstoječih uporabniških vmesnikih, nadaljuje pa s pripravo integriranega uporabniškega vmesnika. V tem koraku je treba formalne opise sestavnih delov pretvoriti v končni videz

uporabniškega vmesnika, vključno z mehanizmi za ravnanje z dogodki in validacijo podatkov. Za izvedbo omenjene faze potrebujemo model, s katerim bomo definirali arhitekturo osrednje aplikacije, ki bo skrbela za komunikacijo med končnim uporabniškim vmesnikom in integriranimi uporabniškimi vmesniki. V primeru spletnih uporabniških vmesnikov bo komunikacija potekala neposredno s spletnim strežnikom. Osrednja aplikacija bo prav tako vsebovala logiko za izgradnjo končnega uporabniškega vmesnika in njegov videz.

2.2.5 Primerjava integracije na nivoju uporabniških vmesnikov z ostalimi

Glavna razlika integracije na nivoju uporabniškega vmesnika v primerjavi z ostalimi integracijami je v tem, da je tipično nadzorovana s pomočjo dogodkov, posredno s strani uporabniških akcij [39]. V primeru uporabniške akcije se mora komponenta namreč ustrezno odzvati in spremeniti svoje stanje, ob tem pa morajo preostale komponente, ki sestavljajo uporabniški vmesnik, to spremembo zaznati ter se ustrezno prilagoditi trenutnemu stanju. Tako vedno dosežemo konsistentno stanje uporabniškega vmesnika, kar zagotavlja njegovo pravilno delovanje.

Pri uporabniških vmesnikih je pomembna tudi stopnja nadzora komponent. Obstajajo komponente, ki jih lahko nadziramo grafično, v tem primeru gre običajno za namizne aplikacije. Nadzor nad tovrstno komponento praktično ni možen, razen prek pozicije kazalca miške, kar pa zelo oteži razvoj vmesnika. Nad izbrano komponento lahko izvajamo nadzor tudi prek vmesnikov API, ki so lahko nizkonivojski in omogočajo kontrolo nad posameznimi elementi komponente, ali pa visokonivojski, kjer z določenim ukazom ali entiteto spremenimo obnašanje komponente.

Dodatno težavo predstavlja arhitektura uporabniškega vmesnika, ki je v primerjavi z ostalimi nivoji veliko bolj zaprta in predstavlja zaključeno celoto. V primeru spletnih uporabniških vmesnikov se komponente ne zavedajo sosednjih komponent, zato komunikacija med njimi poteka izključno prek strežnika, ki upravlja vse informacije spletnega uporabniškega vmesnika. Strežnik v tem primeru predstavlja edino točko za dostop do podatkov, torej uporabnik nima neposredno dostopa do aplikacije, kjer je poslovna logika.

Na nivoju poslovne logike obstajajo mehanizmi za komunikacijo med moduli, ki pokrivajo določeni sklop funkcionalnosti. V tem primeru je lažje vzpostaviti nadzor nad moduli in tako spremljati dogodke, na podlagi katerih se lahko preostali moduli ustrezno odzovejo. Na nivoju uporabniških vmesnikov takšnega mehanizma ni moč zaslediti. Medtem ko bi v primeru namiznih uporabniških vmesnikov lahko implementirali kanal za dostop do komponent, pa je v primeru spletnih uporabniških vmesnikov to praktično nemogoče. Težavo predstavlja strežnik, ki ima v tem primeru popoln nadzor nad uporabniškim vmesnikom in se mu je nemogoče izogniti.

2.3 Izzivi integracije spletnih uporabniških vmesnikov

Spletni uporabniški vmesniki sodijo v družino skritih vmesnikov. Za njih je značilno, da ima komponenta sicer možnost nadzora z uporabo vmesnikov, vendar so ti zelo slabo ali pa sploh niso specificirani. Pri spletnih aplikacijah predstavlja takšen vmesnik kar strežnik sam, saj mu lahko prek zahtevka HTTP (*Hypertext Transfer Protocol*) pošljemo ukaz za spremembo, le da v tem primeru ne moremo predvideti, kakšen bo končen videz uporabniškega vmesnika. Prikaz uporabniškega vmesnika bo temeljil na odgovoru strežnika, zato so takšni vmesniki glede na varnost in arhitekturo relativno nezanesljivi. S tem smo identificirali prvi in tudi glavni problem integracije spletnih uporabniških vmesnikov, in sicer kako določiti podatkovni model komponente na uporabniškem vmesniku in kako nadzirati njeno obnašanje.

Tako kot pri konceptu EAI (*Enterprise Application Integration*) je tudi pri integraciji uporabniških vmesnikov treba definirati komponente, ki bodo skupno na koncu predstavljale rezultat integracije. Pri integraciji na nivoju podatkovnih baz se v ta namen običajno uporabijo poizvedbe SQL (*Structured Query Language*), ki so definirane na bazi in jim drugače pravimo tudi pogledi. Na aplikacijskem nivoju pa se za potrebe prenosa podatkov uporabljajo označevalni jeziki, različice jezika XML (*Extensible Markup Language*), za definicijo programskih vmesnikov pa naprednejši opisni jeziki, kot je WSDL (*Web Service Description Language*). Povezovanje programskih vmesnikov s poslovno logiko temelji na osnovnem programskem jeziku, najbolj znan je Java, lahko pa gre za bolj specifičen jezik, kot je to storitveno-kompozitni jezik WS-BPEL (*Web Services Business Process Execution Language*). Tako integracija na podatkovnem nivoju kot tudi integracija na aplikacijskem nivoju uporabljata dokaj zrele kompozitne jezike, v nasprotju z integracijo na nivoju uporabniških vmesnikov, ki še čaka na pravo rešitev [24]. Torej je v ta namen treba definirati kompozitni jezik, s pomočjo katerega bodo naslovljeni različni parametri in komponente za integracijo na nivoju uporabniških vmesnikov.

Pomemben del integracije uporabniških vmesnikov je tudi vizualen videz, kjer pa se postavlja vprašanje, kdo naj izvede vizualizacijo, komponenta sama ali aplikacija, ki ji pripada. Dejansko je problem v tem, ali naj komponenta sama poskrbi za svojo vizualizacijo ali pa naj aplikacija prek identifikatorja komponente pripravi njen izris. Za identifikatorje se uporablja označevalni jezik, in sicer je ta lahko namenjen izključno dokumentom, med katere sodi večina označevalnih jezikov, ki temeljijo na jeziku XML, ali pa je namenjen označevanju uporabniškega vmesnika, kot na primer XAML (*Extensible Application Markup Language*). Ne glede na tip označevalnega jezika za interpretacijo tega potrebujemo svoje orodje (npr. brskalnik), ki definicije spremeni v grafične elemente. Tovrstne definicije tipično vsebujejo le statični del elementov, za odzivnost in dinamiko uporabniškega vmesnika pa so na voljo skriptni jeziki, med katere sodi skriptni jezik *JavaScript*, ki je podprt v večini brskalnikov. Če povzamemo, imamo dva načina izrisa komponent na uporabniški vmesnik, in sicer izris,

nadzorovan od komponente, to je običajno v namiznih aplikacijah, in izris s pomočjo identifikatorjev, kjer komponenta posreduje identifikator aplikaciji, ki nato poskrbi za lokacijo in videz komponente.

Za ključni del integracije pa je treba pripraviti tudi ustrezen komunikacijski protokol, s pomočjo katerega bodo komponente na uporabniškem vmesniku komunicirale med seboj in z aplikacijo, ki skrbi za integracijo [24]. Komponente morajo prepoznati vsako uporabniško akcijo na vmesniku in to ustrezno sporočiti svoji nadrejeni aplikaciji, ki pa mora na podlagi dogodka pripraviti odgovor, če je potrebno, pa ukaz za vsako izmed komponent. Pri integraciji na nivoju podatkovnih baz je medsebojna komunikacija enostavnejša, saj imamo na eni strani podatkovne baze, ki predstavljajo pasivne udeležence, in na drugi strani centralno enoto, ki skrbi za izvajanje ter potek dogodkov. Podobno je pri integraciji na aplikacijskem nivoju. Tako imamo v našem primeru lahko centralizirano komunikacijo, kjer centralna enota sprejema in proži dogodke, ali pa neposredno komunikacijo med komponentami, kjer ima integracijska aplikacija le vlogo ene izmed komponent, vpletenih v izvajanje dogodkov. Komunikacijo lahko izvedemo na način RPC (*Remote Procedure Call*) s pomočjo klicev metod, ali pa s pomočjo dostavljanja sporočil, kjer se izbrana komponenta lahko naroči na posamezno temo (*topic*) za prejemanje sporočil in jih prav tako tudi objavlja.

Povezovanje komponent s centralno aplikacijo, ki izvaja samo integracijo, sodi v množico večjih izzivov, ki jih moramo nasloviti. Glavni težavi pri povezovanju sta, kako identificirati komponente in kako pridobiti njihovo referenco, prek katere nato poteka komunikacija. To lahko storimo na dva načina, in sicer statično ob načrtovanju ali pa dinamično med izvajanjem aplikacije, običajno prek vnaprej definiranih registrov [24]. Pri integraciji na nivoju podatkovnih baz se povezovanje podatkovnih virov definira med izvajanjem aplikacije, in sicer s pomočjo že vnaprej določene centralne podatkovne sheme, ki definira, kateri podatki se bodo črpali in iz katerih virov. Podobno je z integracijo na aplikacijskem nivoju. Čeprav je tehnično dopustno uporabiti dinamično raziskovanje komponent in njihovo povezovanje, pa takšen način ni vedno dobrodošel, saj je v primeru novih komponent izvajanje interakcije bistveno težje. Tako je večinoma v uporabi hibriden način povezovanja, kjer izbrano množico komponent identificiramo že v fazi načrtovanja, določen del teh pa med izvajanjem. Za hibriden način povezovanja je značilno, da je definiranje komponent opredeljeno že ob načrtovanju, pridobivanje njihovih referenc pa v fazi izvajanja aplikacije.

2.4 Pregled obstoječih rešitev

Tako kot pri ostalih področjih imamo tudi v našem primeru že obstoječe rešitve in tehnologije, ki so nastale z namenom olajšati integracijo aplikacij na nivoju uporabniških vmesnikov. Pri tem moramo ločiti med tehnologijami, ki so usmerjene predvsem v rešitve na področju namiznih aplikacij, kjer je možno delati izključno z grafičnimi komponentami in dogodki. Naša odločitev je bila, da se osredotočimo izključno na spletne tehnologije, zato se

bomo podrobneje seznanili le z rešitvami in tehnologijami, ki so vezane na integracijo spletnih uporabniških vmesnikov.

Za integracijo na nivoju spletnih uporabniških vmesnikov se sicer najde že več predlogov in konceptov [24], a je le redko kateri zasnovan generalno ali pa je ostalo le pri teoriji, brez praktične rešitve. Najboljša osnova za naš primer je prav notacija BPMN (*Business Process Model Notation*) [9], s pomočjo katere so opisani in grafično predstavljeni poslovni procesi podjetja. Pri opisovanju poslovnih procesov lahko naletimo na akcijo, ki jo mora izvesti uporabnik, čemur rečemo uporabniško opravilo. To pomeni, da mora uporabnik prek uporabniškega vmesnika izvesti interakcijo z aplikacijo. Za pripravo uporabniškega vmesnika lahko poskrbi razvijalec, obstaja pa tudi možnost generiranja uporabniških vmesnikov na podlagi obstoječih definicij uporabniških opravil, za kar poskrbijo ogrodja, med katera uvrščamo tudi ogrodje ADF (*Application Development Framework*) [21], razvito v podjetju Oracle. V tem primeru notacija BPMN predstavlja poleg modela tudi obliko zapisa poslovnih procesov, torej ga lahko poimenujemo tudi opisni jezik. Glede na to, da je naš cilj doseči integracijo spletnih uporabniških vmesnikov, kjer bosta uporabniku zagotovljena minimalna verjetnost ponovitve vnosa iste informacije in izboljšana uporabniška izkušnja, za ogrodje ADF tega žal ne moremo z gotovostjo trditi. Prav tako je znano, da generirani uporabniški vmesniki na uporabnika vplivajo negativno, saj se ne da zagotoviti, da bi bil ta všečen uporabniku in prilagojen optimalnemu vnosu podatkov [14].

Omenimo lahko tudi najbolj primitiven približek integracije s pomočjo izsekov kode (*snippets*), ki služijo za izmenjavo informacij z že vnaprej pripravljenim in oblikovanim delom vmesnika. Glavna pomanjkljivost omenjene rešitve pa je statičnost vmesnika, saj ga je zelo težko prilagajati videzu. Če izseki predstavljajo informacije v obliki člankov, ki vključujejo tudi slike ali grafe, pa nastane dodatna težava razporeditve vsebine. Podoben koncept je tudi tehnologija RSS (*Rich Site Summary*), ki prek vmesnika API ponuja povzetke izbrane spletne strani; običajno so vsebinsko razdeljeni na podlagi specifikacije oz. pravil, ki definirajo tehnologijo. Največkrat jo najdemo na spletnih straneh, ki posredujejo dnevne novice končnim uporabnikom. Prednost za tovrstno uporabo je v tem, da je tehnologija dokaj preprosta in znana, zato njena uporaba ne zahteva dodatnega vložka v razvoj, kar pa je dobrodošlo tako s stališča ponudnika kot tudi partnerja, ki želi novice prikazovati v svoji aplikaciji, in sicer s svojim stilom oblikovanja. Slabost pri tem pa je, tako kot pri večini obstoječih rešitev, da je preveč specifična in težko prilagodljiva, poleg tega je tudi zelo omejena, saj jo je težko razširiti za domeno z večjim naborom informacij.

Če preidemo na rešitve, ki so se približale našemu cilju, hkrati pa so še bolj specifično usmerjene kot že prej omenjene tehnologije, potem lahko omenimo še projekt SoKNOS [10]. Nastal je na podlagi ideje, s katero bi olajšali planiranje in upravljanje reševanja v primeru naravnih nesreč večjih razsežnosti. V času načrtovanja in razvoja se je v Nemčiji z njim

ukvarjala skupina inženirjev, in sicer s pomočjo sodelovanja ključnih podjetij oz. društev, ki so vpeta v mrežo zaščite prebivalstva. Rešitev temelji na ontologijah oz. semantičnih tehnologijah, a je še vedno preveč specifična, saj je bil glavni cilj projekta razviti namizno aplikacijo, ki bo s pomočjo integracije uporabniških vmesnikov različnih informacijskih sistemov omogočala hitro in učinkovito koordiniranje reševanja v primeru naravnih nesreč.

2.5 Pregled tehnologij

V tem delu bomo pregledali pristope za integracijo uporabniških vmesnikov, predvsem s stališča spletnih vmesnikov, ki so trenutno pogosto uporabljeni za predstavitev informacij uporabnikom.

2.5.1 Komponente kot vtičniki spletnih brskalnikov

Začetki integracije spletnih uporabniških vmesnikov segajo v čase, ko je bilo predvajanje večpredstavnostnih vsebin na spletu še znanstvena fantastika, vendar je to pomenilo začetek drugačnega razmišljanja. Animacije so na spletnih uporabniških vmesnikih dobivale svoj prostor in so postale vedno bolj priljubljene. V takšnih primerih so razvijalci največkrat uporabili spletne tehnologije, kot so skriptni jezik JavaScript in vgrajene komponente, imenovane vtičniki.

Komunikacija z vtičniki temelji na zelo preprostem zunanjem vmesniku, ki običajno zahteva le nabor določenih nastavitev, katerih vrednosti mora razvijalec nastaviti ob vgradnji, v označevalno kodo XHTML (*Extensible Hypertext Markup Language*), ki v tem primeru predstavlja integracijski jezik. Vgrajene komponente zagotavljajo predstavitev vsebine s pomočjo lastnega interpreterja, običajno pa omogočajo tudi komunikacijo med komponento in preostalim delom spletne strani, v katero je komponenta vgrajena. Tehnologija z vgrajenimi komponentami je načeloma lahka za uporabo, največjo težavo pa je v začetku predstavljalo pomanjkanje ogrodij za komunikacijo med komponentami. Običajno so razvijalci omenjeno težavo odpravljali tako, da so za komunikacijo med komponentami uporabili napredno programsko kodo, napisano v jeziku JavaScript, kar pa je bilo v nasprotju s poenotenim pristopom za delo s komponentami. Spletni brskalniki, ki skrbijo za izvajanje vtičnikov, predstavljajo večjo omejitev kot pa model za integracijo komponent.

2.5.2 Hibridi spletnih storitev

Med bolj priljubljenimi so tudi hibridi spletnih storitev (*web mashups*), ki oblikujejo vsebino portalov ali spletnih strani tretjih ponudnikov. Običajno uporabljajo za te namene pripravljene vmesnike API. V začetku pojava tovrstnih storitev vmesniki API niso bili zanesljivi, saj se ponudniki vsebine največkrat sploh niso zavedeli, da se vsebina njihove spletne strani prikazuje tudi na tujih portalih, kar je v osnovi pomenilo zlorabo teh vmesnikov. V zadnjem času se je pojavilo nekaj poskusov reševanja tovrstne komunikacije, in sicer s

pomočjo ontologij in semantike [5]. Ideja je, da razvijalec označi vse elemente posameznega vmesnika API, in sicer glede na vlogo elementa. Elemente je treba razlikovati med kategorijami, operacijami in dogodki. Na podlagi označb nato orodje vzpostavi register semantičnih opisov, s pomočjo katerih razvijalcu kasneje lahko pripravi tudi programsko kodo spletnega uporabniškega vmesnika. Največjo oviro pri tovrstnih aplikacijah predstavlja prav komunikacija med komponentami, zato je največkrat rezultat le statični prikaz vsebine, brez možnosti interakcije. Tudi v primeru razvoja tovrstnih vmesnikov je večji del razvojnega časa treba nameniti ročnemu testiranju, saj so vgrajene komponente izredno nezanesljive. Ugotovili smo, da je izgradnja hibridov spletnih storitev težka in časovno potratna naloga.

2.5.3 Spletni portali s komponentami

Pristop za razvoj spletnih portalov razlikuje med komponentami uporabniškega vmesnika (*web portlets*) in integracijsko aplikacijo (*web portal*) ter v osnovi predstavlja enega izmed naprednejših pristopov za integracijo uporabniških vmesnikov. Komponente, iz katerih so sestavljeni spletni portali, se pojavljajo v obliki vtičnikov, ki imajo nalogo generiranja delov portala v obliki programske kode XHTML. Pri tem je treba upoštevati nekatera pravila za oblikovanje portala, vključno s postavitvijo posameznega dela. S tem razvijalci dosežejo skladnost s preostalimi komponentami, in sicer s stališča oblikovanja spletnih uporabniških vmesnikov. Spletni portali imajo še eno dobro lastnost; uporabnikom namreč dovoljujejo spremembo videza končnega uporabniškega vmesnika. Tako lahko uporabniki razvrščajo, onemogočajo in omogočajo posamezne komponente portala in si s tem zagotovijo osebno prilagojen videz portala. Ta prednost pa posredno zahteva obvezno identificiranje uporabnika v aplikaciji, kar je pri spletnih portalih običajno že ena od osnovnih zahtev. Večina portalov namreč vključuje tako spletni forum za diskusijo kot tudi borzne tečajnice in ostale dnevne informacije. Tovrstne komponente portala želijo uporabniku informacije čim bolj približati, za kar potrebujejo tudi določene informacije o uporabniku.

Razvoj komponent temelji na implementaciji specifičnih vmesnikov, ki izhajajo iz definicije za standardni spletni vmesnik API. Razvijalcem komponent je tako omogočeno, da pripravijo komponento, ki jo bo moč vključiti v katerokoli standardno ogrodje za gradnjo spletnih portalov. Spletni portal je v končni fazi predstavljen s pomočjo združenih delov programske kode XHTML, ki pa so rezultat vključenih komponent. Jedro portala mora skrbeti tudi za medsebojno komunikacijo integriranih komponent. Opisani pristop omogoča tako statično kot tudi dinamično povezavo med komponentami. Komunikacija med komponentami na podlagi skupnih podatkovnih modelov povzroča večjo odvisnost, zato je priporočljiva uporaba skupnega mehanizma za komuniciranje, kot na primer sporočilno vodilo (*Message Bus*) [16]. Čeprav imajo komponente, ki so pripravljene s pomočjo različnih programskih jezikov, skupne cilje in skupno arhitekturo, pa vseeno med seboj največkrat niso neposredno povezljive. Prav s tem namenom je nastala tehnologija WSRP (*Web Services for Remote*

Portlets) [18, 37], ki skrbi za samodejno komuniciranje med portali na nivoju spletnih storitev. Prednost omenjene tehnologije je v tem, da razvijalcu ni treba skrbeti za komunikacijo s storitvami, temveč tehnologija s pomočjo opisnega jezika UDDI (*Universal Description, Discovery and Integration*) [36] sama najde opis izbrane storitve, na podlagi katerega vzpostavi komunikacijo z izbrano storitvijo. Na takšen način so spletni portali med seboj lažje povezljivi, s tem pa je tudi čas razvoja primerno krajši.

2.5.4 Razčlenitev spletnega uporabniškega vmesnika

Glede na to, da je večina spletnih uporabniških vmesnikov razvitih brez tipske osnove in s pomočjo najbolj priljubljenih spletnih tehnologij, kot so XHTML, CSS (*Cascading Style Sheets*) in JavaScript, bomo preverili tudi možnost integracije na tem področju. V takšnem primeru imamo uporabniški vmesnik, za katerega pa ne poznamo strukture, niti vsebovanih elementov. Tako moramo uporabiti pristop od zgoraj navzdol (*top-down*) in definirati metodologijo, s katero bomo lahko na najbolj pameten način identificirali ter označili posamezne elemente uporabniškega vmesnika, njihovo vlogo in njihove odvisnosti ter povezave med njimi.

Nastalo je nekaj konceptov, ki uporabljajo različne pristope in različne tehnologije. Med njimi smo zasledili poskus integracije s pomočjo spletnih jezikov UML (*Unified Modeling Language*) in WebML (*Web Modeling Language*), za modeliranje [35], kjer bi uporabili obstoječi pristop generiranja uporabniških vmesnikov na podlagi spletnega modeliranja. Integracija bi tako temeljila na obstoječih tehnologijah za modeliranje spletnih uporabniških vmesnikov. V tem primeru je treba v koncept modeliranja umestiti dodatno plast, na kateri se bo izvajala integracija vmesnikov. To pomeni, da je treba vhodne podatke (modele) preoblikovati v model, na podlagi katerega bo generiran končni spletni uporabniški vmesnik. Generiranje programske kode temelji na konceptu MDE (*Model Driven Engineering*) [35], ki s pomočjo definiranih transformacij preoblikuje vhodne podatke v pričakovane rezultate. Tovrstna integracija ima sicer potencial, vendar je ostalo le pri prototipu, za dejansko uporabo pa bi bilo treba odpraviti še nekaj vrzeli. Ena izmed večjih je izguba informacij pri generiranju programske kode uporabniškega vmesnika.

Podoben pristop je nastal s pomočjo ontologij in semantičnih tehnologij, kjer koncept predvideva definiranje ontologij za vsakega izmed uporabniških vmesnikov [24]. Te ontologije pa morajo pripadati vnaprej predvideni množici konceptov določenega področja. Med njimi veljajo določena pravila in obstajajo relacije ter medsebojne odvisnosti. Za definiranje ontologij sicer obstaja več tehnologij, med bolj znanimi so RDF Schema (*Resource Description Framework*), OWL (*Web Ontology Language*) [22] in F-Logic (*Frame Logic*). Z ontologijami je torej opisan nabor pravil, na podlagi katerih poteka členitev spletnega uporabniškega vmesnika. Uporabniški vmesnik je sestavljen iz množice elementov, ki jih je treba identificirati glede na definirano množico ontologij. Temu pravimo semantično notiranje, s pomočjo

katerega nastaja baza podatkov, ki v našem primeru služi za integracijo uporabniškega vmesnika. S pomočjo semantike namreč opredelimo element glede na vlogo in pomen, ki ga ima, kar se kasneje odraža z boljšim razumevanjem prenesenih informacij med komponentami. Integracija pa je izvedena s pomočjo ovijanja (*encapsulating*) aplikacije v vtičnik [4], ki skrbi za prikaz uporabniškega vmesnika in izmenjavo dogodkov med posameznimi vtičniki. Nad semantičnimi podatki sicer lahko izvajamo tudi poizvedbe, vendar le z jeziki za semantične poizvedbe, med katere sodi SPARQL (*SPARQL Protocol and RDF Query Language*). Ontologije so v tem primeru osnova za vzpostavitev baze podatkov, s katerimi je moč manipulirati. Omenjena tehnologija se torej vede zelo podobno kot običajne podatkovne baze, le da v tem primeru ontologije določajo strukturo podatkov.

2.6 Pomanjkljivosti obstoječih tehnologij

Če povzamemo trenutne rešitve za integracijo spletnih uporabniških vmesnikov, ki so na voljo, so večinoma zasnovane zelo specifično in so zaradi tega težko prilagodljive. Nekatere izmed njih so namenjene izključno izbranemu pristopu. Lahko bi rekli, da so le za enkratno uporabo ali pa za uporabo v primeru zelo podobnih situacij, tako vsebinsko kot problemsko opredeljenih. Če je rešitev zasnovana za širši spekter uporabe, torej za različne primere uporabe, pa je velikokrat težava združljivost med različnimi tehnologijami. Tako so nekatere rešitve namenjene izključno integraciji namiznih aplikacij, kjer grafične komponente zahtevajo popolnoma drugačen pristop kot pa komponente za spletno uporabo. Dodatno težavo pri obstoječih rešitvah povzroča tudi menjava problemske domene, saj večkrat zahteva preveč prilagajanja in s tem povezane komplikacije. Že za prilagoditve je potreben dodaten čas, ki ga razvijalci največkrat ne načrtujemo, saj se zanašamo na že obstoječe komponente.

S stališča neodvisnosti tehnologij imamo pri obstoječih rešitvah prav tako določene omejitve. Obstoječi koncepti oz. njihovi standardi ne predvidevajo uporabe novejših tehnologij, zato so večinoma neskladni z njimi, razen v primeru, ko se standard vztrajno posodablja in s tem usklajuje z novejšimi tehnologijami. Pri večini rešitev se uporabljajo obstoječe osnove, kot v primeru jezika, s pomočjo katerega je moč označiti ključne dele aplikacije, ki so potrebni za delovanje. Tovrstnih jezikov je mnogo, med bolj znanimi je XML, ki je največkrat tudi osnova za ostale različice. Za namen definiranja uporabniških vmesnikov je bilo zasnovanih več jezikov, med opaznejšimi so WebML, ki je namenjen grafičnemu modeliranju spletnih aplikacij na višjem nivoju, IFML (*Interaction Flow Modeling Language*) za grafični prikaz interakcije uporabnika z vmesnikom, DiaMODL (*Dialog Model Language*) [31], ki je namenjen za modeliranje dialogov, MARIA XML (*Model-based Language For Interactive Applications*) [23] za modeliranje interaktivnih aplikacij v vseprisotnih okoljih in UsiXML (*User Interface Extensible Markup Language*) [17, 19], namenjen opisovanju uporabniških vmesnikov za uporabo v različnih kontekstih (*znakovni*, *grafični*, *slušni* in *več-modalni*). Omenjeni jeziki so

namenjeni uporabi na višjem nivoju, predvsem za predstavitev modela, nekateri pa niti niso namenjeni spletnim uporabniškim vmesnikom.

Med opisnimi jeziki sta se našim potrebam najbolj približala XIML (Ex*tensible Interface Markup Language*) in XPIL (Ex*tensible Presentation Integration Language*). Jezik XIML [38] je namenjen opisu uporabniških vmesnikov z namenom gradnje na različnih platformah, ki jih uporabljajo naprave. Zasnovan je torej tako, da se razvijalec lahko elegantno izogne problemu podpore različnih platform. Tehnologija sama poskrbi za prilagojeno gradnjo uporabniškega vmesnika, glede na platformo naprave, s katero uporabnik dostopa do aplikacije. Jezik omogoča tudi opisovanje grafičnih elementov za izris, saj je v osnovi namenjen predstavitvenim uporabniškim vmesnikom in ne zajemu podatkov. Pri tem pa je jezik XPIL dejansko namenjen integraciji spletnih uporabniških vmesnikov, vendar za izvedbo integracija uporablja brskalnik [39], v katerem so tako osrednja aplikacija kot tudi vsi adapterji, ki so potrebni za komunikacijo s spletnimi uporabniškimi vmesniki. Ta model temelji torej na integraciji spletnih uporabniških vmesnikov na strani odjemalca.

Ugotovimo lahko, da so nekatere rešitve oz. pristopi zelo togi, prilagajanje novim tehnologijam pa je praktično nemogoče. Na tem mestu lahko zatrdimo, da trenutno ne obstaja rešitev za integracijo spletnih uporabniških vmesnikov na strani strežnika, ki bi bila neodvisna od tehnologije.

3 Model za integracijo spletnih uporabniških vmesnikov

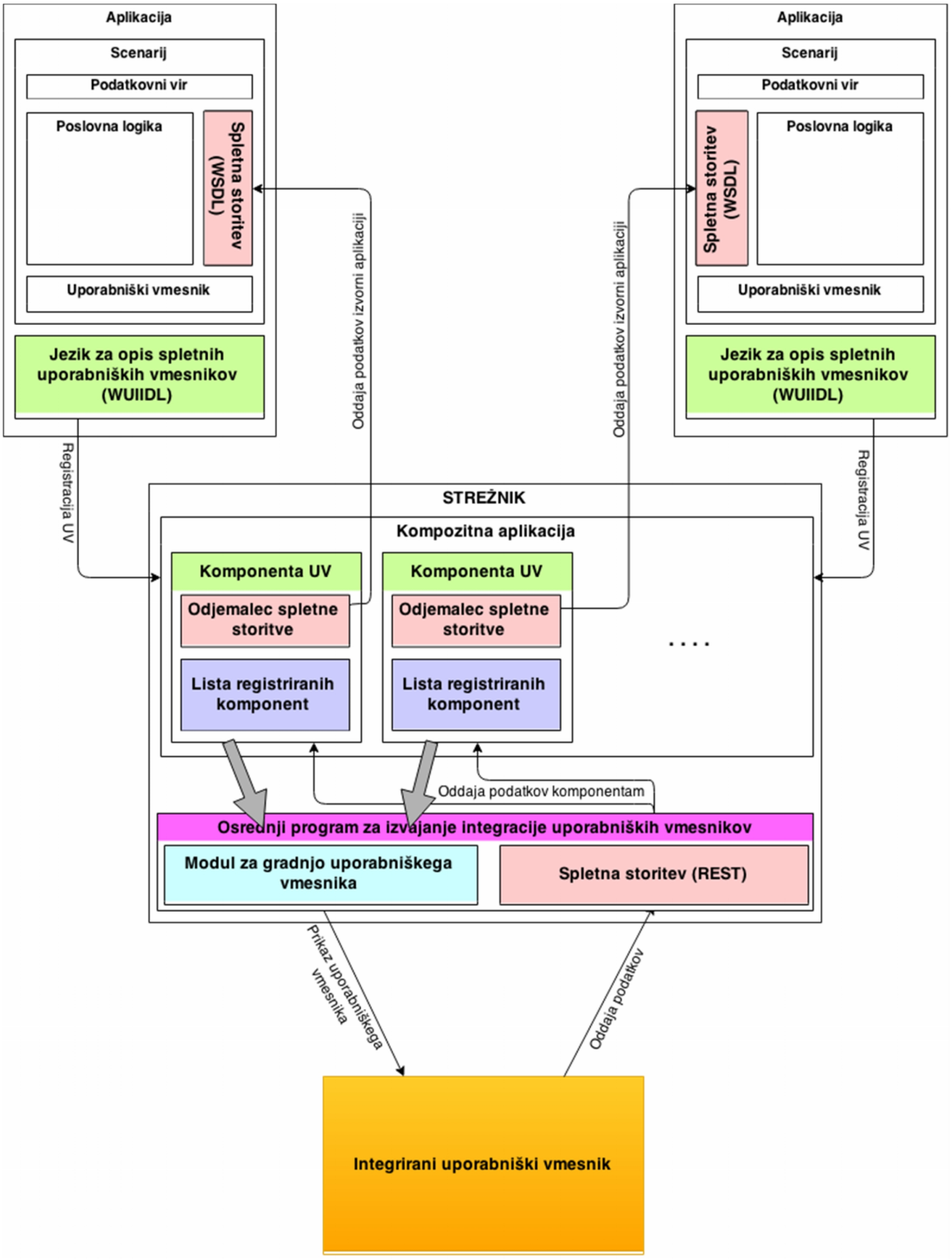
Naš cilj je definirati model za integracijo spletnih uporabniških vmesnikov na strani strežnika, zato se osredotočimo izključno na spletne tehnologije in spletne uporabniške vmesnike, ki so sestavni del spletnih aplikacij. Osrednji del magistrskega dela tako predstavlja razvoj modela za integracijo spletnih aplikacij na nivoju uporabniških vmesnikov, pri čemer je koncept neodvisen od tehnologij. Z uporabo tega si bo razvijalec poenostavil razvoj spletnega uporabniškega vmesnika, s čimer bo razvoj spletnih aplikacij postal učinkovitejši, uporabniki pa se bodo izognili ponavljajočim se vnosom podatkov za doseganje svojega cilja.

Zasnovali smo model za integracijo spletnih uporabniških vmesnikov, ki skrbi tako za registracijo uporabniških vmesnikov kot tudi za komunikacijo med njimi in na koncu za generiranje spletnih uporabniških vmesnikov. Ključni del modela je specifično v ta namen zasnovan jezik za opis integracijskih točk, ki jih potrebujemo za izvajanje integracije. Jezik smo poimenovali WUIIDL. Z njim bomo opisali stične točke integracije in s tem zagotovili komunikacijski protokol.

3.1 Arhitektura modela

Model smo arhitekturno razdelili na več delov, in sicer na osrednjo aplikacijo (*middleware*), komunikacijski del skupaj z opisnim jezikom WUIIDL in končni uporabniški vmesnik, ki predstavlja rezultat integracije. Osrednja aplikacija skrbi za postopek integracije aplikacij in njihovih vmesnikov, zaveda pa se tudi vseh komponent, ki so vključene v integracijo. Komponente se shranjujejo v ločenem registru, s pomočjo katerega osrednja aplikacija izvaja integracijo vmesnikov in pripadajočih komponent. Model za izvedbo integracije potrebuje vhodne podatke, ki so predstavljeni kot opisi uporabniških vmesnikov in pripadajo izbranim aplikacijam SaaS (*Software as a Service*). Uporabniški vmesniki so opisani z uporabo opisnega jezika WUIIDL, ki je bil razvit v okviru magistrskega dela. Rezultat integracije uporabniških vmesnikov se odraža kot uporabniški vmesnik, ki se je generiral na podlagi vhodnih podatkov in ustreza pogojem ter izpolnjuje vse dane naloge. Ta sicer predstavlja edino točko interakcije med uporabnikom in aplikacijo, zato ima v modelu zelo pomembno vlogo.

Za izvedbo postopka integracije potrebujemo modul, s katerim bomo izvajali združevanje posameznih komponent, prisotnih v uporabniških vmesnikih. V končni fazi osrednja aplikacija, skupaj s preostalimi deli za izvajanje integracije predstavlja glavno aplikacijo, ki teče na strežniku, in obenem odgovarja na zahtevke uporabnikov oz. njihovih odjemalcev. Aplikacija odgovarja na zahtevke v obliki zgrajenega spletnega uporabniškega vmesnika, kar predstavlja rezultat integracije.



Slika 3.1: Arhitektura modela.

Osrednjo aplikacijo smo razdelili na več manjših delov (Slika 3.1), in sicer komunikacijski del, registracija uporabniških vmesnikov skupaj s komponentami, integracijski del, modul za pripravo uporabniškega vmesnika ter modul za sprejem podatkov. S tem smo zagotovili lažjo obvladljivost in prilagodljivost modela. Vsak izmed delov je v aplikacijo umeščen s posebnim

namenom, zaradi katerega je bil tudi razvit. Tako so posamezni deli modela popolnoma neodvisni in šibko sklopljeni, s čimer posledično dosežemo lažje vzdrževanje in večjo učinkovitost. Tako smo del modela, ki je namenjen izključno upravljanju z vhodnimi podatki, torej branju in razumevanju vhodnih informacij ter hranjenju informacij o uporabniških vmesnikih, ločili od dela, kjer poteka priprava na integracijo uporabniških vmesnikov. Integracija vmesnikov prav tako poteka v svojem delu modela, podobno velja za izvajanje komunikacije z obstoječimi aplikacijami, s pomočjo katere bomo dostavljali informacije na cilj. Model predvideva možnost komuniciranja z zunanjim svetom le s pomočjo vnaprej določenih točk, ki predstavljajo kanale za izvajanje komunikacije.

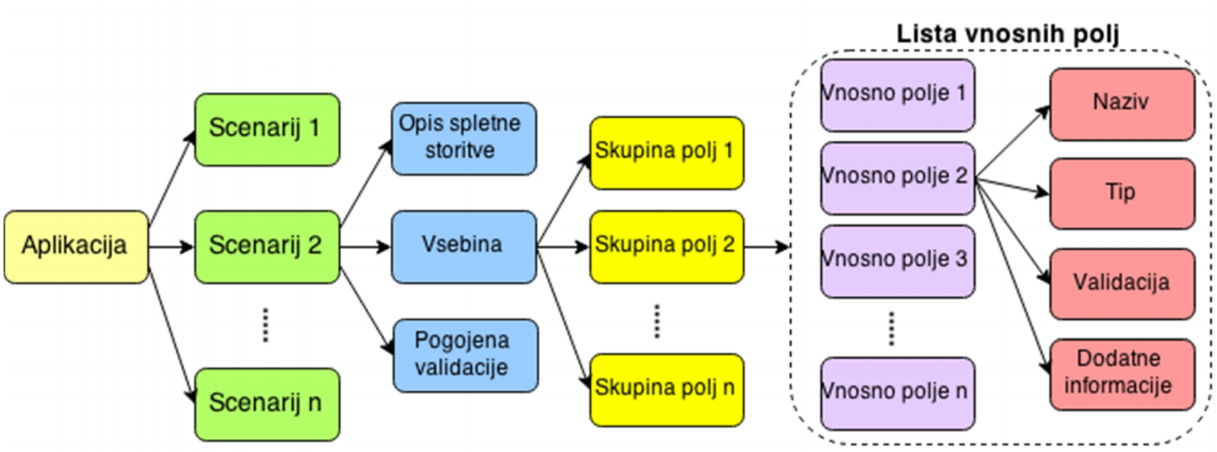
Od vsake aplikacije, katere uporabniški vmesnik želimo integrirati, pričakujemo, da ima izpostavljena vsaj dva vmesnika, s pomočjo katerih bomo lahko izvedli integracijo. Potrebujemo torej vmesnik za dostop do datoteke z opisi pripadajočih spletnih uporabniških vmesnikov, s pomočjo katerih bomo lahko pridobili vse informacije, potrebne za pripravo končnega spletnega uporabniškega vmesnika. Drugi tip vmesnika pa je spletna storitev, prek katere bomo lahko komunicirali z aplikacijo oz. njeno poslovno logiko, in sicer glede na potrebe dinamične validacije, za dostop do seznama dovoljenih vrednosti (šifranti) in pa na koncu za oddajo podatkov, ki jih bo uporabnik vnesel v spletni uporabniški vmesnik. Ti vmesniki predstavljajo stične točke integracije. Poslovna logika in hranjenje podatkov sta torej neodvisna, kar je tudi cilj integracije spletnih uporabniških vmesnikov.

3.2 Jezik za opis integracijskih točk (WUIIDL)

Za začetek smo pozornost usmerili v razvoj jezika za opis integracijskih točk, ki jih bo aplikacija kasneje prepoznala in vključila v postopek integracije. Najprej smo naslovili množico zahtev, ki jih mora izpolnjevati nastali jezik. S tem smo prišli do naslednjega nabora:

- opis posameznih vmesnikov, ki so definirani na podlagi obstoječih aplikacij;
- opis pravil za izvedbo validacije posameznih komponent na vmesniku;
- opis protokolov in parametrov za komunikacijo med aplikacijami;
- združevanje komponent na končnem uporabniškem vmesniku v smiselne celote.

Na podlagi zahtev smo izbrali tehnologijo, s pomočjo katere smo definirali strukturo opisnega jezika. Za osnovno tehnologijo smo uporabili označevalni jezik XML, za katerega smo pripravili definicijo strukture (Slika 3.2), in sicer z uporabo shem XSD (*XML Schema Definition*).



Slika 3.2: Struktura opisnega jezika WUIIDL.

Definicijo strukture opisnega jezika smo na nivoju spletnega uporabniškega vmesnika razdelili na več delov:

- skupina parametrov za opis vmesnika,
- skupine komponent vmesnika,
- skupina parametrov za opis komunikacije,
- skupina parametrov za validacijo informacij.

Pri definiciji posameznih vmesnikov smo se še posebej osredotočili na komponente, ki sestavljajo uporabniški vmesnik. Omeniti je treba tudi skupino parametrov za opis pravil validacije informacij. Parametri so v osnovi sicer vezani na komponente, ki so vključene v vmesnik, zato smo jih vključili v opis komponent spletnega vmesnika. Vrhnji element končnega opisa predstavlja aplikacijo, ki jo želimo integrirati, vsebuje pa množico komponent, ki so opisane v ločenem delu modela.

3.2.1 Parametri za opis vmesnika

V shemo za opis spletnega uporabniškega vmesnika smo vključili tako splošne informacije vmesnika kot tudi parametre za opis komunikacijskega protokola in zbirko informacij za opis komponent, ki predstavljajo spletni vmesnik. V tem delu bomo pozornost namenili predvsem splošnim in tehničnim informacijam vmesnika, kamor sodita naslov ter naziv vmesnika, s pomočjo katerega bomo kasneje v aplikaciji lahko ločevali med vmesniki. Ta del opisa je torej namenjen izključno identifikaciji opisanega vmesnika s pomočjo informacij, ki se nanašajo le nanj.

3.2.2 Komponente vmesnika

Ključni del spletnega uporabniškega vmesnika predstavljajo komponente, s pomočjo katerih uporabniki izvajajo interakcijo z aplikacijo. Za opis komponent smo v poglavju 3.7.1 definirali shemo, ki zajema:

- naziv komponente,
- tip komponente,
- opis komponente (pomoč ali dodatna informacija),
- opis pravil za validacijo informacij, poslanih od komponente, in
- možnost dodatnih informacij o komponenti (omejitev možnosti interakcije).

Naziv komponente potrebujemo predvsem za njeno identifikacijo, tako s tehnične strani kot tudi s strani uporabnika, saj ima lahko naziv komponente za uporabnika poseben pomen. Kot pomoč uporabniku smo namenili poseben parameter, s pomočjo katerega lahko z dodatnim pojasnilom uporabniku pomagamo pri interakciji s komponento. Vsebuje lahko informacijo o tem, kakšen tip informacije je pričakovan od uporabnika, obliko informacije ali pa navsezadnje, čemu je sploh namenjena komponenta. Vsaka informacija ima določen tip in obliko zapisa, kjer mislimo predvsem na ločitev med opisnimi informacijami komponente ter informacijami o vsebini komponente. Postopek preverjanja pravilnosti informacije bomo natančneje opisali v poglavju 3.4, zato bi v tem delu omenili le to, da je preverjanje lahko vezano izključno na eno ali pa na določen nabor komponent spletnega uporabniškega vmesnika, in sicer je s tem podana zahteva po medsebojni odvisnosti komponent. Pri komponentah imamo opcijo uporabniku ponuditi že vnaprej definiran nabor možnosti interakcije, ki jih lahko prikažemo na vmesniku, in sicer s pomočjo grafičnih omejitev (poglavje 3.3), kot je to v primeru vnosnih polj, kjer imamo na voljo seznam (*dropdown*) z opcijskimi vrednostmi ali pa listo potrditvenih polj oz. radijskih gumbov. V ta namen smo definirali dodaten parameter za podporo vnaprej omejenih vrednosti komponente. S tem smo tudi zaključili z opisom komponent na spletnih uporabniških vmesnikih.

3.2.3 Parametri za opis komunikacije

Omenili smo že, da je pri integraciji zelo pomembna tudi komunikacija med komponentami. V našem modelu potrebujemo komunikacijo na nivoju spletnih vmesnikov, saj je informacije, poslane od komponent, treba na koncu dostaviti na cilj, torej določeni aplikaciji, pred tem pa tudi ustrezno preveriti. Preverjanje lahko izvedemo s statično validacijo, kjer je informacija veljavna že s podanim vzorcem, kateremu mora ustrezati, lahko pa to storimo s pomočjo dinamične validacije, za katero pa potrebujemo dodatno orodje za izvajanje komunikacije s spletno storitvijo. Za klic spletne storitve potrebujemo točno definiran protokol in končni

naslov, kje je storitev na voljo. Za opis komunikacije smo sicer definirali tudi lastno strukturo, ki smo jo podrobneje opisali v poglavju 3.7.5. Opazimo lahko, da je komunikacija pomemben del modela, saj brez nje izvedba ni možna.

3.2.4 Parametri za opis validacije informacij

Tako kot ostali deli opisa vmesnika je tudi validacija pomemben sklop integracije spletnih uporabniških vmesnikov. Končnemu uporabniku aplikacije nikoli ne smemo popolnoma zaupati, saj lahko hitro zaide iz okvirov uporabe vmesnika, lahko le zaradi nerodnosti, obstaja pa tudi možnost zlorabe. Parametre za opis validacije smo zasnovali po konceptu množice opisov pravil (poglavje 3.4.1), s katero lahko za eno samo komponento predpišemo več različnih pravil za validacijo informacij. Vsako preverjanje informacije lahko opišemo s pomočjo tipičnih pravil in dodatnih parametrov, ki jih zahteva izbrani tip pravila. S tem smo zagotovili izvajanje validacije, ki je neodvisna od preostalih komponent. Za validacijo, ki je medsebojno odvisna, pa smo pripravili posebno definicijo strukture (poglavje 3.7.5), ki je vezana na celoten spletni uporabniški vmesnik, in vsebuje informacije o dostopnosti validacije kot spletne storitve. Model namreč ne predvideva količine in vrste informacij, ki jih pričakuje, zato je brez takšnih informacij težko zagotoviti ustrezen postopek preverjanja vhodnih informacij. Tako smo odgovornost za tovrstne postopke prenesli na izvorno aplikacijo, in sicer s pomočjo spletne storitve, ki mora zagotavljati preverjanje medsebojne odvisnosti posredovanih informacij.

3.3 Komponente vmesnika

Pri postopku integracije spletnih uporabniških vmesnikov imajo glavno vlogo prav komponente, ki v končni fazi predstavljajo videz uporabniškega vmesnika. Komponente predstavljajo glavno vez med uporabnikom in uporabniškim vmesnikom ter hkrati dovoljujejo interakcijo med uporabnikom in vmesnikom.

V integracijskem modelu smo komponente obravnavali kot najbolj občutljive sestavne dele modela, saj jih je na vsakem koraku postopka integracije treba pravilno procesirati. Z njihovo pomočjo je namreč možen edini stik z uporabnikom, in če bi prišlo do izgube informacij, bi to kasneje povzročilo posledice, kot so napake v nadaljnjem poteku dogodkov. Poznamo več vrst komponent, ki zahtevajo različen tip interakcije z uporabnikom, ki jim tudi omejuje nabor možnosti in območje gibanja. Ločimo med komponentami, ki so namenjene prostemu vnosu informacij, in komponentami, ki so namenjene le interakciji z uporabnikom in po videzu delujejo bolj dinamično. Mednje sodijo komponenta za izbiro podstrani (*menu*) (Slika 3.3), komponenta za združevanje preostalih komponent v zaključene celote (*panel*), komponenta za osredotočeno komuniciranje z uporabnikom (*dialog*) in komponenta za boljši pregled informacij (*table*).

Slika 3.3: Primer komponente za izbiro podstrani.

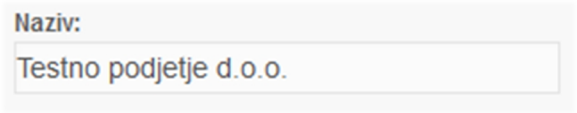
Izvedba večine omenjenih komponent je zelo odvisna od njihovega pomena in načina uporabe. Tako lahko komponento za osredotočeno komuniciranje (*dialog*) razvijalec prilagodi svojim potrebam, lahko ji namreč doda gumbe, s katerimi uporabniku ponudi možnost izbire njegovega odziva. V tem primeru lahko govorimo o uporabi osnovne komponente s funkcionalnimi prilagoditvami, s pomočjo katerih lahko razvijalec izboljša uporabniško izkušnjo.

Več pozornosti moramo nameniti komponentam za zajem in posredovanje informacij, kjer sta pomembna tudi tip in oblika informacije. V osnovi jih razlikujemo glede na njihovo uporabo, in sicer:

- za vnos informacije v obliki niza (*textarea*);
- za izbiro informacije s pomočjo spustnega seznama (*dropdown*);
- za izbiro informacije s pomočjo radijskih gumbov (*radio button*);
- za izbiro informacije s pomočjo potrditvenih polj (*checkbox*);
- za predstavitev grafičnih elementov (*image*);
- za potrditev podatkov na uporabniškem vmesniku (*button*).

3.3.1 Komponenta za vnos informacije v obliki niza

Komponente za vnos informacij v obliki znakovnega ali številskega niza so najbolj pogosto uporabljene komponente za zajem informacij. Poznamo več različic omenjene komponente, ki so prilagojene glede na zahtevano obliko niza. To je lahko daljši tekst (opis nečesa) ali pa znakovni oz. številski niz z določenimi omejitvami, na katere mora biti uporabnik pri vnosu informacije še posebej pozoren. Komponente so sicer podobne podolgovatemu okencu (Slika 3.4), kamor uporabnik lahko zapiše podatke, ki se ob potrditvi vmesnika kot informacija z vsebino pošlje na strežnik, ta pa jo mora ustrezno obravnavati glede na trenuten kontekst.

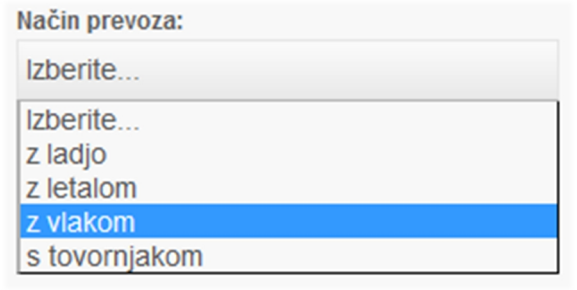


Slika 3.4: Komponenta za vnos niza.

Za lažjo uporabo je komponenta običajno opremljena z dodatnimi oznakami, ki uporabniku nudijo lažjo predstavo o tem, kakšen podatek je treba vnesti v komponento. Te oznake so lahko v obliki krajšega statičnega teksta ali pa dodatnega, malo daljšega opisa. Za prepoznavanje napak, ki so bile identificirane v postopku izvajanja validacije, pa skrbi posebej za ta namen opremljena komponenta za prikaz opisa napake, ki se običajno pojavi v stilu izstopajočih komponent, saj naj bi jo uporabnik opazil kar najhitreje.

3.3.2 Komponenta za izbiro informacije s pomočjo spustnega seznama

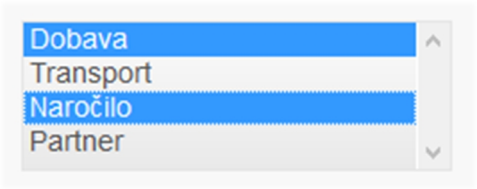
Komponente, ki uporabniku omejujejo nabor veljavnih informacij, so opremljene z dodatnim spustnim seznamom, na katerem se prikazuje vnaprej definiran nabor informacij. Komponenta na pogled ne deluje nič drugače kot komponenta za vnos informacij v obliki niza, le da je v tem primeru opremljena še s spustnim seznamom (Slika 3.5) in nanj običajno ni možno vpisati poljubnega podatka. Ima tudi dodatno omejitev, in sicer dopušča izbiro le ene izmed vrednosti na spustnem seznamu.



Slika 3.5: Komponenta s spustnim seznamom.

V poslovnih aplikacijah se omenjena komponenta pojavlja zelo pogosto, saj je enostavna za vgradnjo v uporabniški vmesnik, poleg tega je le enovrstična in tako ne zaseda preveliko prostora. Od slabih lastnosti komponente je treba omeniti naslednji dve. Nabor vrednosti na spustnem seznamu mora biti primerno omejen, sicer postane seznam nepregleden. Proces validacije informacije pa mora biti izveden v sodelovanju z registrom, v katerega smo shranili prikazani nabor vrednosti v seznamu. Kljub vsem omejitvam, ki jih komponenta omogoča, pa tudi v primeru njene uporabe ne smemo popolnoma zaupati uporabniku. Pri spletnih tehnologijah se moramo namreč zavedati, da v verigi nastopa veliko tehnologij, ki so podprte z različnimi orodji, in skoraj pri vsakem izmed njih obstaja možnost zlorabe. Prav zaradi tega moramo biti pri sprejemu informacij še posebej previdni.

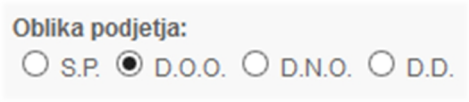
Obstaja tudi izpeljanka pravkar opisane komponente, in sicer lista z vnaprej definiranimi vrednostmi, ki so prikazane kot običajen seznam (Slika 3.6), v katerem pa je možno izbrati več vrednosti hkrati. Komponenta v takšni obliki zavzame več prostora na uporabniškem vmesniku. Uporabna je v primerih, ko uporabniku želimo ponuditi možnost izbire več opcij hkrati, in sicer skupaj s hitrim pregledom vseh vrednosti.



Slika 3.6: Komponenta s prikazanim seznamom.

3.3.3 Komponenta za izbiro informacije s pomočjo radijskih gumbov

Podobno kot komponenta s spustnim seznamom tudi komponenta za izbiro informacije s pomočjo radijskih gumbov uporabnika omejuje na izbiro le ene izmed vseh naštetih opcij. Razlika je v predstavitvi nabora informacij, in sicer so radijski gumbi (Slika 3.7) običajno razporejeni na izbranem območju, kjer uporabnik lahko opazuje celoten nabor hkrati.

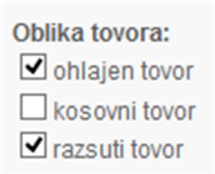


Slika 3.7: Komponenta z radijskimi gumbi.

Za izbiro informacije uporabniku ni treba izvajati posebne interakcije, to stori šele po tem, ko sprejme končno odločitev. Komponenta je zelo uporabna v primerih, ko uporabniku želimo pokazati zelo omejen nabor informacij, ki se med seboj izključujejo, torej lahko izbere le eno izmed njih. Slabost komponente je otežena vgradnja v uporabniški vmesnik, še posebej v primeru, ko se seznam možnosti spreminja, in je velikost območja, ki ga zaseda komponenta, treba dinamično prilagajati.

3.3.4 Komponenta za izbiro informacije s pomočjo potrjevalnih polj

Ta je pomembna, če želimo od uporabnika doseči, da med vnaprej pripravljenimi možnostmi izbere svoj nabor, za katerega meni, da je najbolj primeren za odgovor na zastavljeno vprašanje. Prikaz komponente je zelo podoben prikazu komponente za izbiro informacije s pomočjo radijskih gumbov, razlikuje se le v tem, da se možnosti med seboj ne izključujejo (Slika 3.8), temveč je možno izbrati podmnožico ponujenih opcij.



Slika 3.8: Komponenta s potrjevalnimi polji.

Ob izboru možnosti se pred njo izriše potrditvena kljukica, ki na koncu predstavlja končni izbor uporabnika. S ponovnim klikom na možnost se ta odstrani. Tudi v tem primeru mora biti nabor opcij primeren, v nasprotnem primeru bo uporabniški vmesnik postal nepregleden.

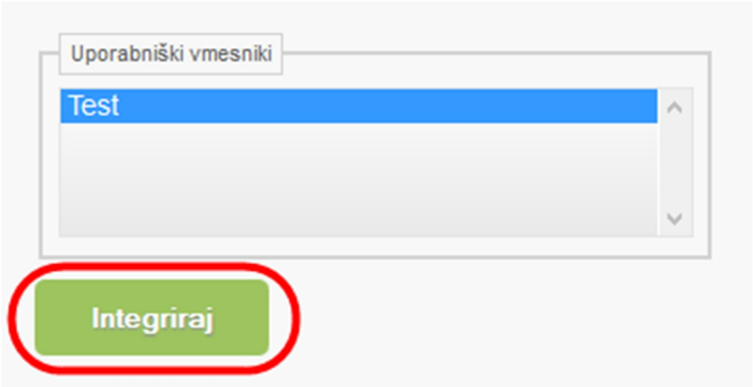
3.3.5 Komponenta za predstavitev grafičnih elementov

Kot na vsakem drugem uporabniškem vmesniku, je treba tudi pri spletnih vmesnikih prikazati izbrane grafične elemente, kot so slike, grafi, vzorci in ostalo gradivo, s pomočjo katerega se uporabniku poskuša čim bolj poenostaviti predstavitev vsebine vmesnika. Glavna vloga komponente za prikaz tovrstnih elementov je posredovanje informacij o lokaciji grafičnega elementa in njegove predvidene velikosti odjemalcu. Obstajajo pa različni načini uporabe komponente, in sicer jo razvijalci dostikrat zlorabijo za prikaz stilsko dovršenega gumba (Slika 3.9), ki ima osnovno nalogo izvedbo akcije.

Slika 3.9: Primer stilsko dovršenega gumba s sliko (*vir: www.facebook.com*).

3.3.6 Komponenta za potrjevanje podatkov na uporabniškem vmesniku

Uporabniški vmesniki so sestavni del spletnih poslovnih aplikacij. Za izvajanje interakcije med uporabniškim vmesnikom in uporabnikom pa potrebujemo dodatno komponento, ki jo imenujemo gumb za potrjevanje (Slika 3.10).



Slika 3.10: Primer uporabe komponente za potrjevanje podatkov.

Osnovna naloga gumba je proženje akcije za procesiranje podatkov uporabniškega vmesnika. Če pri procesu validacije pride do napak, se običajno ponovno prikaže uporabniški vmesnik, in sicer z vključenimi opisi napak. Ob tem poznamo tudi posebno različico gumba, ki je nastala z namenom uporabe v korist boljši uporabniški interakciji. To pomeni, da torej za delovanje ne potrebuje spletnega obrazca z izpolnjenimi podatki, temveč služi za splošne namene, med katere sodita tudi izbiranje podstrani v meniju in odpiranje dialogov.

3.4 Validacija

Po spoznavanju osnovnih komponent na spletnih uporabniških vmesnikih moramo opisati še koncept izvajanja validacije informacij. Omenili smo že, da moramo ločiti med dvema tipoma validacije:

- statična validacija posameznih informacij;
- dinamična validacija z medsebojno odvisnostjo med informacijami.

Oba tipa validacije skrbita za veljavnost informacij, ki prihajajo v obdelavo od uporabniškega vmesnika. S tem zagotavljamo, da ne bi prišlo do neskladnosti podatkov v podatkovnih bazah ali pa v aplikaciji, ki bi se od tistega trenutka dalje začela obnašati neobičajno. Skladnost informacij ima v primeru modela za integracijo uporabniških vmesnikov poseben pomen, saj moramo zagotoviti, da bodo vse informacije, ki jih bomo prejeli od uporabniškega vmesnika, skladne z vsemi pravili, ki jih integrirane aplikacije implementirajo. Model za integracijo in množica vseh integriranih aplikacij se morata obnašati homogeno, sicer bi v trenutku prišlo do razkola v procesiranju informacij in bi vse interakcije z uporabnikom postale neveljavne.

3.4.1 Statična validacija

Enostavnejša validacija informacij pomeni, da nimamo opravka z medsebojnim preverjanjem informacij, od katerih bi bila informacija v trenutku validacije odvisna. Takšni validaciji lahko rečemo tudi statična validacija. Z njo običajno preverjamo veljavnost informacije s pomočjo naslednjih pravil:

- obveznost podatka,
- znakovni ali numerični podatek,
- dolžina znakovnega podatka (minimalna in maksimalna),
- preverjanje vrednosti numeričnega podatka (maksimalna in minimalna),
- vzorčno preverjanje znakovnega podatka (*regular expression*) in
- nabor vrednosti podatka (manjše množice statičnih vrednosti).

3.4.1.1 Obveznost podatka

V primeru, ko od uporabnika želimo v vsakem primeru pridobiti izbrano informacijo in je izpolnitev komponente obvezna, govorimo o preverjanju obstoja podatka. To pravilo validacije je eno enostavnejših, saj lahko podatek obstaja ali pa ga ni. Težavo pri preverjanju obveznosti podatka predstavljajo očem nevidni znaki, kot so presledek, prehod v novo vrsto in zamik, zato jim pravimo tudi nevidni oz. kontrolni znaki. Uporabnik lahko v komponento vnese zgolj zaporedje nevidnih znakov, kaj storiti v tem primeru, saj je podatek prisoten, vendar iz njega ni moč izluščiti informacije oz. vsebine.

3.4.1.2 Znakovni ali numerični podatek

Uporabniki so lahko zelo nepredvidljivi, zato so lahko informacije, ki prihajajo od uporabniškega vmesnika, neskladne z zahtevanimi. To pomeni, da je treba preverjati tudi tip poslanega podatka. Največkrat želimo ločiti med znakovnimi in numeričnimi podatki, saj se nadaljnji postopki validacije razlikujejo za vsakega izmed njiju. Pri spletnih tehnologijah je uporabniku zelo težko zaupati v tolikšni meri, da bi bile informacije podane v takšni obliki, kot jo predvideva komponenta.

3.4.1.3 Dolžina znakovnega podatka

Znakovne podatke, ki jih uporabnik vnese na vmesniku, je treba preveriti tudi s stališča dolžine, in sicer bi lahko prevelika dolžina podatka povzročila težavo ob zapisovanju podatka v podatkovno bazo. V ta namen uporabljamo validacijo s preverjanjem dolžine podatka, in sicer imamo možnost omejitve dolžine tako navzgor kot tudi navzdol. Od uporabnika lahko torej zahtevamo podatek, ki je omejen tako z minimalno kot tudi maksimalno dolžino.

3.4.1.4 Preverjanje vrednosti numeričnega podatka

Podobno kot pri omejevanju znakovnih podatkov imamo tudi pri numeričnih podatkih možnost omejitve njihove vrednosti. Tako lahko določimo spodnjo mejo, pod katero vrednost podatka ne sme priti. Enako velja za zgornjo mejo. Takšno omejitev lahko konkretno uporabimo v primeru podatka o starosti osebe, saj je naravno znano, da človek težko doseže več kot 100 let, razen z določenimi izjemami, zato lahko nastavimo zgornjo mejo na 150 let. Prav tako lahko nastavimo spodnjo mejo starosti na vrednost 0 let, če zahtevamo, da so osebe dopolnile polnoletno starost, pa lahko to mejo seveda prilagodimo, glede na predpise v izbrani državi. S tem smo uporabniku omejili vnos nesmiselne vrednosti, ki bi imela za posledico negativne učinke pri nadaljnjih korakih.

3.4.1.5 Vzorčno preverjanje znakovnega podatka

Za določene vrste podatkov lahko preverjamo tudi strukturo podatka, s čimer zagotavljamo, da je uporabnik v primeru podobnih podatkov vpisal pravilen podatek v pravilno komponento. Tako lahko kot primer navedemo podatek o e-poštnem naslovu, za katerega obstaja določen vzorec (*pattern*), ki predpisuje dovoljeno strukturo e-naslovov. Vzorčno preverjanje sicer

največkrat udejanjimo s pomočjo tehnologije Regular Expression, ki jo podpira večina visokonivojskih programskih jezikov.

3.4.1.6 Nabor vrednosti podatka

Ta velja v primeru podatka, ki smo ga na uporabniškem vmesniku predstavili z vnaprej definiranim naborom opcij, za katerega velja, da so popolnoma neodvisne in statične, torej se v času izvajanja ne spreminjajo. V takšni situaciji lahko prav tako uporabimo statično validacijo, in sicer v obliki preverjanja končnih vrednosti podatka s pomočjo množice vrednosti. Primer takšne validacije je preverjanje podatka o spolu osebe, kjer imamo samo dve možnosti, in sicer moški ali ženski spol. S tem zagotovimo, da je uporabnik resnično izbral eno izmed vnaprej ponujenih možnosti, in se tako izognemo nadaljnjim težavam.

3.4.2 Dinamična validacija

Dinamična validacija je v nasprotju s statično bolj kompleksna, saj predstavlja preverjanje kompleksnejših pravil, ki so običajno zapisana v poslovnih zahtevah. Tako imamo lahko dinamičen register z naborom vrednosti ali pa se nabor veljavnih vrednosti spreminja glede na kontekst, v katerem je trenutno uporabnik. Obstaja tudi možnost, da se nabor veljavnih vrednosti spreminja glede na vrednosti referenčnega podatka, torej gre za medsebojno odvisnost dveh podatkov. Zato je tovrstno validacijo razmeroma težko implementirati in ima za posledico kompleksnejše izvajanje celotne aplikacije.

Za dinamično validacijo sicer obstajajo posebne tehnologije, ki so namenjene prav preverjanju skladnosti in ustreznosti podatkov, glede na kontekst, v katerem so. Tako poznamo validacijo podatkov s pomočjo semantičnih tehnologij [1], med katere sodi tudi ogrodje RDF [27], ki služi opisovanju pravil in medsebojne odvisnosti podatkov. Semantični splet je v zadnjem času ena izmed pomembnejših vej spletne informatike in prav tako ena izmed hitreje razvijajočih se tehnologij.

V našem modelu smo se odločili, da z namenom zagotavljanja skladnosti informacij podpremo dinamično validacijo, ki jih pričakujejo integrirane aplikacije. Zaradi prevelike kompleksnosti in velikega nabora pravil smo tovrstno preverjanje prepustili pripadajočim aplikacijam, in sicer bomo uporabili v ta namen pripravljene spletne storitve. Zato smo v modelu definirali tudi posebno strukturo za izvajanje validacije s pomočjo dinamičnih pravil, ki ji lahko nastavimo parametre za komuniciranje s spletno storitvijo.

Pri izvajanju validacije imajo posebno vlogo tudi sporočila (Slika 3.11), s katerimi uporabniku sporočimo, kje na uporabniškem vmesniku so nastali problemi oz. kateri podatek ne ustreza danemu pravilu.

Slika 3.11: Povratna sporočila validacije.

Ta sporočila so pri statični validaciji bolj enostavna od sporočil pri dinamični, saj morajo vsebovati informacijo o dovoljenih vrednostih ali pa celo o dovoljenih kombinacijah. Sporočila morajo vsebovati toliko informacij, da bo uporabnik lahko z njimi rešil nastalo situacijo na uporabniškem vmesniku.

3.5 Komunikacija s spletnimi storitvami

Omenili smo že, da je komunikacija med komponentami pomembnejši del integracije, saj bi bila izvedba modela brez nje otežena. Za komunikacijo se v informatiki sicer uporabljajo različne tehnologije, ki so že nekaj časa bolj ali manj znane. Omenili bomo le dve najbolj razširjeni:

- protokol SOAP (*Simple Object Access Protocol*) in
- arhitekturni stil REST (*Representational State Transfer*).

SOAP je bil med prvimi protokoli, ki so bili namenjeni komunikaciji s spletnimi storitvami [28], med drugim pa z vključevanjem razširitev zagotavlja tako varnost (*WS-Security*) kot tudi zanesljivost (*WS-ReliableMessaging*) storitve komuniciranja. Uporaba protokola SOAP spada med zahtevnejše, če ne potrebujemo njegovih prednosti, pa postane tudi neučinkovit. Zaradi

tega je nastal arhitekturni stil REST [4], ki je v zadnjem času zaradi svoje enostavnosti postal še posebej razširjen.

3.5.1 Struktura za opis spletne storitve (WSDL 1.1)

Opisni jezik WSDL 1.1 predstavlja shema, s pomočjo katere predpišemo pravila za komunikacijo med odjemalcem in ponudnikom spletne storitve. Strukturo jezika WSDL lahko razdelimo na več delov, in sicer je glavni del prav opis storitve (*service*). V tem delu opisa moramo definirati ime spletne storitve, ki bo vidno navzven in bo tako predstavljalo spletno storitev. Storitvi z uporabo vezave (*binding*) dodelimo vrata (*port type*), s pomočjo katerih posredno definiramo še operacije (*operation*) spletne storitve. Za vsako vezavo moramo prav tako definirati različne parametre, med katerimi je tudi izbira protokola za komuniciranje. Za vsako operacijo moramo definirati tudi sporočila (*message*), ki predstavljajo vhodne in izhodne parametre za izbrano operacijo. Sporočilo je v končni fazi predstavljeno z določeno strukturo XML, ki je lahko definirana znotraj datoteke WSDL, ali pa je definirana v zunanji, ločeni shemi. Vsi deli skupaj tvorijo opis spletne storitve, ki jo lahko uporabimo tako za predstavitev kot tudi za implementacijo spletne storitve.

3.5.2 Ponudnik spletne storitve

Tako kot pri ostalih spletnih tehnologijah moramo tudi za spletno storitev najprej priskrbeti ustrezno lokacijo, na kateri bo spletna storitev na voljo v vsakem trenutku. Vsaka spletna storitev mora imeti na voljo svojo predstavitev storitve, in sicer v smislu, katere operacije so na voljo in kakšna je struktura vhodnih oz. izhodnih parametrov. Najbolj razširjen in tudi najbolj znan opisni jezik, ki podpira definiranje spletnih storitev, je prav jezik WSDL [34]. Dinamičnost jezika omogoča tako razširjeno uporabo, obstaja pa tudi zadostna količina orodij za razvoj, ki podpirajo jezik WSDL. Pri tem ni odveč omeniti, da so za lažjo uporabo razvili tudi orodje, s katerim lahko na podlagi opisa v jeziku WSDL generiramo programsko kodo, ki služi kot osnova za implementacijo spletne storitve. Enako velja na strani odjemalca, kjer potrebujemo implementacijo za izvajanje klicev spletne storitve. Tako lahko rečemo, da je opis spletne storitve ključnega pomena, saj služi za centralizirano objavo spletne storitve na spletu. Prav zato je opis storitve največkrat tudi javno objavljen, in sicer na spletnem naslovu, ki se običajno konča s končnico »?wsdl«, kot na primer »*http://server:8080/ServiceName/OperationName?wsdl*«. Pri tem lahko opazimo, da imamo v eni sami spletni storitvi lahko na voljo več operacij, ki določajo vsebino izvajanja spletne storitve in različne parametre. Jezik WSDL se največkrat pojavlja v kombinaciji s protokolom SOAP, ki definira protokol za izvedbo komunikacije in tako zagotavlja tudi ustrezno zanesljivost izmenjave podatkov.

3.5.3 Odjemalec spletne storitve

Za uporabo spletne storitve potrebujemo odjemalca, ki mora biti implementiran na podlagi opisa WSDL, ki opisuje spletno storitev, sicer medsebojna komunikacija ni mogoča. Za vzpostavitev stika s spletno storitvijo potrebujemo spletni naslov, kjer je opis spletne storitve. Na takšen način zagotovimo, da komuniciramo s točno izbrano storitvijo. Vzpostavitev komunikacije je možno doseči tako z vnaprej pripravljeno implementacijo operacije na izbrani storitvi ali pa dinamično s pripravo zahtevka s pomočjo generičnega ogrodja storitve in protokola SOAP, ki je sicer že integriran v implementaciji. Odjemalec je lahko implementiran v katerikoli tehnologiji, ki podpira komunikacijo prek protokola TCP/IP (*Transmission Control Protocol/Internet Protocol*), torej v večini tehnologij. Odjemalec se mora tako zavedati opisa spletne storitve, saj bi sicer nanjo naslavljal podatke, ki bi bili strukturirani v obliki, ki je spletni storitvi neznana, in bi bili zahtevki neveljavni. Pri komunikaciji odjemalca in ponudnika spletnih storitev je pomembna tudi sinhronost klicev. Klici so tako lahko asinhroni, kar pomeni, da se izvajajo neodvisno od ostalih v ozadju, medtem pa se lahko prožijo že naslednji klici. Drugače je pri sinhronih klicih, kjer je treba počakati na konec izvajanja spodbujene aktivnosti in se šele nato lahko nadaljuje z izvajanjem ostalih aktivnosti. To lastnost lahko izkoristimo za boljšo odzivnost aplikacije oz. uporabniškega vmesnika in posledično tudi za boljšo izkušnjo uporabnika.

V našem modelu potrebujemo komponento za izvajanje klicev spletnih storitev že med izvajanjem validacije in prav tako na koncu za izvajanje procesiranja informacij v okolju izvornih aplikacij. V tem delu modela ima komponenta še pomembnejšo vlogo, saj je od nje odvisna uspešnost procesiranja in posredno tudi informacija, ki jo bo uporabnik prejel na uporabniškem vmesniku. Komponenta za komunikacijo s spletno storitvijo ima torej veliko odgovornost za uspešnost procesiranja informacij. Zato mora biti v tem segmentu modula arhitektura zasnovana čim bolj učinkovito. Vsako neželeno stanje pa je treba ustrezno zabeležiti za kasnejše reševanje morebitnih težav. Pri tem pa ne smemo pozabiti na uporabnika, ki bo s svojo uporabniško izkušnjo zaznamoval uspešnost rešitve.

3.6 Dinamičen prikaz komponent na spletnem uporabniškem vmesniku

Za integracijo spletnih uporabniških vmesnikov potrebujemo tudi način za dinamičen prikaz komponent na vmesniku. V ta namen smo v integracijskem jeziku WUIIDL definirali shemo, ki podpira vsebnik (*container*), ki lahko vsebuje različno število komponent ali pa gnezdenih vsebnikov. Tako dosežemo dinamičnost opisa, ki hkrati podpira širok nabor možnih kombinacij. Opisni jezik torej podpira tovrstne strukture, vendar pa pri tem ne smemo spregledati, kakšne so možnosti udejanjanja tovrstnega opisa uporabniškega vmesnika.

Dinamični prikaz komponent na takšnem vmesniku pa ni tako enostaven kot opis tega. Poleg ostalih omejitev imamo v našem primeru tudi vpliv omejitev s strani spletnih tehnologij,

kjer je dinamično prikazovanje komponent eno zahtevnejših opravil. Omejitev nastane predvsem zaradi spletnih tehnologij, in sicer sta dinamično prikazovanje vmesnika in hkrati všečnost uporabniku zelo razdvojena pojma, ki v tem primeru ne sodita v isti koš. Oblikovanje uporabniških vmesnikov, ki so v celoti pripravljeni v dinamičnem okolju, pomeni oblikovanje neznanega, saj v trenutku razvoja ne moremo natančno vedeti, koliko bo komponent in kako bo vmesnik videti na koncu. Predvideti vse možne scenarije pa je dejansko nemogoče, saj opis vmesnika vsebuje možnost opisa liste komponent, kar pomeni, da je število teh navzgor neomejeno.

3.6.1 Oblikovanje in postavitev dinamično prikazanih komponent

Z dinamičnim generiranjem uporabniškega vmesnika smo poskrbeli za obstoj komponente na vmesniku, poskrbeti je treba še za njen položaj, obliko in odzivnost. Delno za to poskrbijo že nekatera ogrodja, med katerimi je tudi prej omenjeno ogrodje Apache Wicket, delno pa je stvar prepuščena razvijalcu. Pri tem mislimo predvsem na položaj in obliko komponente, kar je sicer lažje kot položaj, saj ne vemo točno, v kakšnem kontekstu je uporabljena. V ta namen si je treba pripraviti nabor možnih stanj. Tako je lahko komponenta le del liste vseh komponent, lahko je uporabljena kot posamična komponenta, lahko pa pripada obstoječi komponenti. V takšnih primerih je izredno težko določiti natančen položaj komponente, še posebno, če ne poznamo njene okolice. Temu se lahko izognemo le s posebnimi označbami komponent, s pomočjo katerih definiramo možne scenarije njihove uporabe. Za listo komponent lahko, na primer, pripravimo celovit vsebnik komponent, ki ga označimo z vnaprej določeno oznako, na katero se lahko sklicujemo pri oblikovanju komponent znotraj omenjenega vsebnika. Podobno lahko storimo za ostale scenarije, vendar se moramo pri tem zavedati, da s tem omejujemo nabor možnih kombinacij postavitve komponent.

Poseben izziv predstavlja vrstni red prikaza komponent. V našem primeru se prikazujejo na podlagi opisa vmesnika, ki predstavljajo glavne vhodne podatke modela. Tako lahko komponenta s povsem enako vsebino in lastnostmi ob vsaki spremembi zamenja svoj položaj, kar predstavlja dodatno oviro za uporabnika, ki si običajno površno zapomni vrstni red komponent oz. gradnikov. Pri integraciji spletnih uporabniških vmesnikov je težko zagotoviti, da bo vsebinsko enaka komponenta vedno na enaki lokaciji oz. vsaj v enakem vrstnem redu, glede na ostale komponente. Na tem mestu potrebujemo poseben algoritem, s katerim bomo lahko ugotovili vsebino komponente, jo označili s posebnimi metapodatki in ji tako določili položaj, glede na vnaprej pripravljeno postavitev uporabniškega vmesnika. Tako ugotovimo, da je težko uskladiti dinamiko vmesnika in uporabniške izkušnje, za katere si želimo, da bi bile čim boljše.

3.6.2 Interakcija uporabnika s prikazanimi komponentami

Za vsako med komponentami na spletnem uporabniškem vmesniku je treba zagotoviti ustrezno komunikacijo s strežnikom, da komunikacija poteka usklajeno in se komponenta temu primerno tudi prilagaja. Za upravljanje dogodkov komponente mora razvijalec poskrbeti sam, in sicer ima na voljo več tehnologij. Mednje sodi tehnologija AJAX (*Asynchronous JavaScript and XML*), ki jo lahko razdelimo na dva dela, in sicer strežniški del, kjer poteka procesiranje dogodka, ter del na strani brskalnika, ki skrbi za pošiljanje dogodkov na strežnik in kasneje za prikaz sprememb na uporabniškem vmesniku. Tehnologija je zelo priročna, saj z njo lahko oskrbujemo le del spletnega uporabniškega vmesnika ali pa samo eno izmed komponent, ki jih vsebuje uporabniški vmesnik, in tako zmanjšamo količino prenesenih podatkov. Primerna je tudi za sprotno validacijo vnosa podatka. To pomeni, da medtem ko uporabnik vnaša podatek, se sproti preverja njegova pravilnost. Obstajajo pa tudi ogrodja za razvoj spletnih uporabniških vmesnikov, ki imajo že vgrajeno podporo za komunikacijo komponent s strežnikom, podati je treba le specifične parametre, na podlagi katerih se nato izvaja proces validacije med vnašanjem podatka. Zelo pomemben del interakcije med uporabnikom in uporabniškim vmesnikom je tudi odzivnost tega, saj smo kot trenutni uporabniki večinoma nepotrpežljivi, tako da je odzivnost vmesnika sestavni del razvoja spletnih uporabniških vmesnikov.

3.7 Definicija strukture opisnega jezika WUIIDL

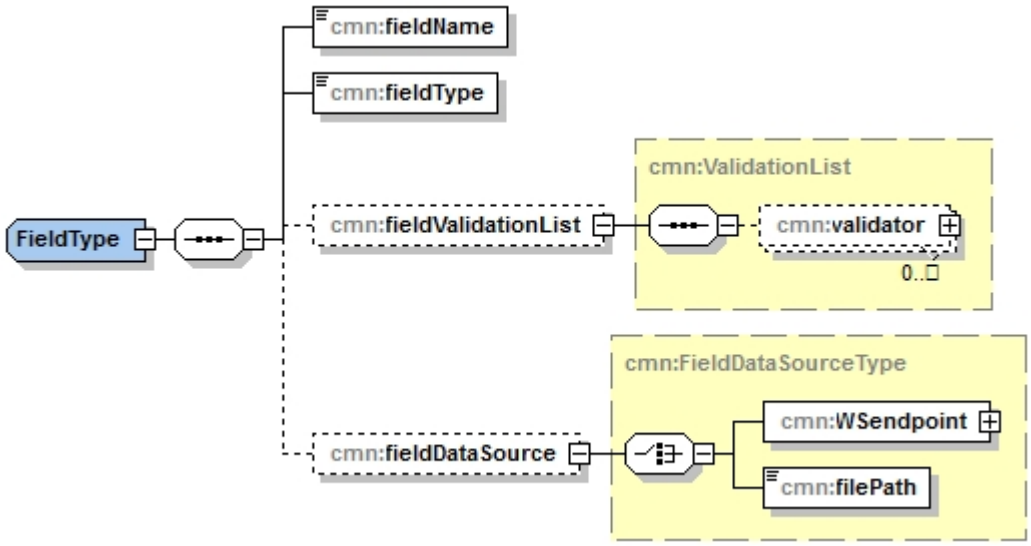
Na podlagi analize spletnih uporabniških vmesnikov smo pripravili sheme, s katerimi smo predpisali oblike struktur za opisovanje skupnih točk spletnih uporabniških vmesnikov. Opis točk smo razdelili na več sklopov, in sicer glede na uporabnosti ter njihov namen. Za uspešno izvedbo integracije spletnih uporabniških vmesnikov namreč potrebujemo informacije o vsebini vmesnika, ki je v primeru prototipa omejena predvsem na vnosna polja vmesnika. Potrebujemo tudi informacije, za dostopnost spletnih storitev, ki so namenjene prav dostavi podatkov s strani integracijskega procesa.

Strukturo jezika XML smo definirali s pomočjo sheme XSD, ki je predstavljala osnovo za nadaljnje delo. Najprej smo pripravili seznam informacij, ki smo jih potrebovali za integracijo vmesnika, in ga primerno razčlenili. Shemo smo ločili na več sklopov, in sicer na sklop z informacijami o vmesniku (obrazcu), sklop z informacijami o spletnih storitvah vmesnika ter sklop z opisi vnosnih polj in pripadajočimi opisi pravil za validacijo posameznega polja.

3.7.1 Struktura *FieldType* za opis vnosnega polja

Za opis vnosnega polja smo za potrebe prototipa definirali kompleksen tip z nazivom *FieldType*. Sestavljen je iz zaporedja elementov (Slika 3.12), med katerimi je tudi element, imenovan *fieldName*, namenjen pa je naslavljanju vnosnega polja. Uporabimo ga lahko tako za potrebe uparjanja vnosnih polj (kar bomo spoznali v naslednjih poglavjih), kot tudi za poimenovanje vnosnih polj na uporabniškem vmesniku. Poleg tega smo vsako vnosno polje

opredelili s stališča oblike vnosne maske, ki so značilne za spletne vmesnike. V ta namen smo definirali poseben nabor možnosti (*FieldTypeEnum*), s katerimi smo omejili dovoljene vrednosti, s čimer si bomo kasneje pomagali pri obravnavi vnosnega polja.

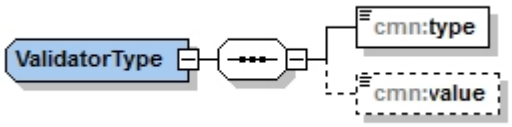


Slika 3.12: Struktura *FieldType*.

Element, ki smo ga izbrali za definiranje informacije o obliki vnosnega polja, smo poimenovali *fieldType*. Če vnosno polje predvideva svoj nabor vrednosti, ki jih lahko ponudi uporabniku, pa je treba tega nekje tudi hraniti. Glede na to, da izbor vrednosti ni v domeni modela za integracijo, smo v ta namen definirali poseben element z nazivom *fieldDataSource*, za katerega smo predvideli opis vira podatkov, ki pa je lahko spletna storitev (*WSEndpointReferenceType*, v poglavju 3.7.5) ali datoteka. Če želimo podatke pred procesiranjem tudi ustrezno preveriti, moramo za potrebe procesa validacije v opis vnosnega polja dodati tudi informacije o pravilih validacije. V ta namen smo dodali element z nazivom *fieldValidationList*, ki predstavlja listo predpisanih pravil za izvajanje validacije. Element se sicer sklicuje na tip *ValidationList*, ki pa v tem primeru predstavlja le ovojnico za lažje označevanje liste elementov kompleksnega tipa *ValidatorType*, ki ga bomo spoznali v naslednjem poglavju.

3.7.2 Struktura *ValidatorType* za opis pravil za izvajanje validacije

S kompleksnim tipom *ValidatorType* (Slika 3.13) smo opisali posamezno pravilo za izvedbo validacije. Za potrebe razvoja prototipa smo vanj vključili element z nazivom *type*, ki je služil predvsem za označitev pravila, zanj pa smo pripravili poseben nabor vrednosti, in sicer s pomočjo enostavnega tipa *ValidatorTypeEnum*.

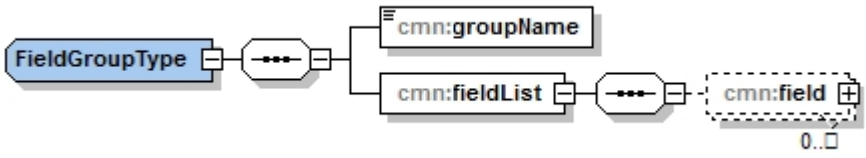


Slika 3.13: Struktura *ValidatorType*.

Poznamo več osnovnih pravil za izvajanje validacije, ki se v večji meri izkažejo za povsem zadostne, včasih pa je treba nabor tudi nadgraditi s kakšnim izmed lastnih pravil. Nekatera pravila za popoln opis potrebujejo dodaten parameter, na primer omejitev vrednosti navzgor, zato smo vgradili dodaten element z nazivom *value*. S tem smo zadostili osnovnim potrebam na področju pravil za izvajanje validacije posameznega vnosnega polja.

3.7.3 Struktura *FieldGroupType* za združevanje vnosnih polj v skupine

Glede na to, da smo se v prototipu osredotočili predvsem na poslovne spletne uporabniške vmesnike, v katerih nastopa večje število komponent oz. vnosnih polj, smo pri tem podprli tudi združevanje komponent v skupine. Tako smo v shemo dodali novo definicijo kompleksnega tipa z nazivom *FieldGroupType* (Slika 3.14), v katerega smo vključili dva elementa, in sicer element *groupName*, ki bo vseboval naziv skupine, ter element *fieldList*, ki predstavlja listo komponent.

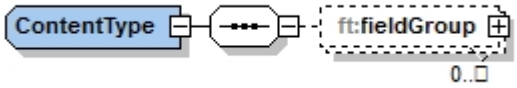


Slika 3.14: Struktura *FieldGroupType*.

Za opis komponent smo se sklicevali na tip z nazivom *FieldType* (poglavje 3.7.1), ki vsebinsko podpira opis posamezne komponente. Pri opisu skupin obstaja možnost razširitve, in sicer bi lahko s pomočjo rekurzivnega elementa dosegli, da bi bilo možno skupine definirati znotraj obstoječe skupine ter s tem posledično zagotoviti zadostno mero dinamike pri opisu vmesnika.

3.7.4 Struktura *ContentType* kot ovojnica za vsebino vmesnika

V opisu vmesnika smo ločili med strukturami za opis metapodatkov vmesnika in elementi, ki nosijo glavne informacije o komponentah vmesnika. Definirali smo ločen kompleksen tip *ContentType* (Slika 3.15), ki služi kot ovojnica vsem opisom vnosnih polj.

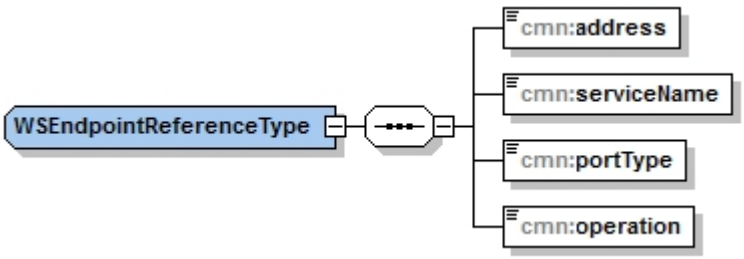


Slika 3.15: Struktura *ContentType*.

V njem smo definirali en sam element *fieldGroup*, ki se sklicuje na že prej omenjeni tip *FieldGroupType* (poglavje 3.7.3), kar nam omogoča morebitno kasnejšo dopolnitev, s katero bi želeli dodatno ločevati med skupinami komponent. Skupin je lahko več, zato smo dopustili možnost ponovitve elementa. S tem smo obdržali celovito strukturo opisanih komponent, v našem primeru vnosnih polj, ki na koncu predstavljajo glavno vsebino spletnega uporabniškega vmesnika.

3.7.5 Struktura *WSEndpointReferenceType* za opis spletne storitve

Za opisovanje informacij o spletnih storitvah vmesnika smo po zgledu koncepta *WS-Addressing* [33] definirali lasten kompleksni tip, ki smo ga poimenovali *WSEndpointReferenceType* (Slika 3.16). Z njim smo posredovali informacije, ki jih potrebujemo za dostop do spletnih storitev.



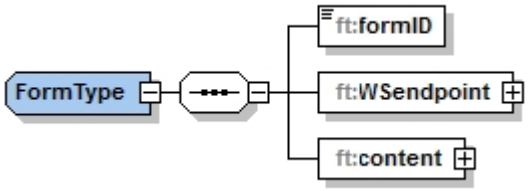
Slika 3.16: Struktura *WSEndpointReferenceType*.

Z njihovo pomočjo bomo namreč izvedli procesiranje podatkov, poslanih od spletnega uporabniškega vmesnika. V kompleksni tip smo vključili element *address*, s pomočjo katerega bomo opisali fizičen naslov spletne storitve. Za ločevanje med posameznimi spletnimi storitvami smo dodali elementa z nazivom *serviceName* in *portType.* Prvi določa naziv spletne storitve, drugi pa vstopno točko do spletne storitve. S pomočjo elementa *operation* bomo označili naziv operacije, s pomočjo katere bomo izvedli klic. Vsaka operacija ima lahko svoje vhodne in izhodne parametre, hkrati pa se od preostalih razlikuje predvsem po implementaciji, torej načinu obdelave podatkov. Tako smo definirali parametre, ki jih bomo uporabili za posredovanje podatkov o spletni storitvi.

3.7.6 Struktura *FormType* za opis celotnega vmesnika

Med zadnjimi smo definirali še kompleksen tip *FormType* (Slika 3.17), namenjen opisu celotnega uporabniškega vmesnika, v osnovi pa predstavlja zaključeno celoto združenih informacij, ki jih potrebujemo za uspešno izvedbo integracije. Vanj smo vključili element

WSendpoint, ki se sklicuje na strukturo *WSEndpointReferenceType* (poglavje 3.7.5) in vsebuje informacije o spletni storitvi. Dodali smo tudi element *content* s strukturo tipa *ContentType* (poglavje 3.7.4), ki pa vsebuje vse potrebne informacije glede vnosnih polj in prav tako pravil za izvajanje validacije. Za potrebe identificiranja vmesnika smo dodali še element z nazivom *formID*. Predstavljen je kot znakovni niz, namenjen pa je izključno unikatni identifikaciji posameznega vmesnika.



Slika 3.17: Struktura *FormType*.

3.7.7 Primer uporabe WUIIDL

Struktura opisnega jezika WUIIDL je definirana, pripravili smo tudi primer njegove uporabe. Uporabniški vmesnik aplikacije tako lahko opišemo v obliki datoteke XML, njegovo pravilnost pa preverimo z uporabo že opisane sheme XSD. Ogledali si bomo primer opisa spletnega uporabniškega vmesnika (Izsek kode 3.1), ki je namenjen vnosu imen navzočih na sestanku. V integracijo bi radi vključili le funkcionalnost za beleženje prisotnih na sestanku, kar pomeni, da potrebujemo polje za vnos imena in opis spletne storitve za vnos podatkov.

Najprej določimo naziv (*formID*) uporabniškega vmesnika, v našem primeru je to *Prisotnost*, s pomočjo katerega ga bomo lahko kasneje identificirali. Za potrebe komunikacije s poslovno logiko aplikacije potrebujemo tudi spletno storitev, ki jo opišemo s pomočjo kompleksnega elementa *WSendpoint*. Omenjena struktura vsebuje elemente, kot so *address*, *serviceName*, *portType* in *operation.* Z elementom *address* določimo naslov za dostop do spletne storitve. Glede na to, da datoteka WSDL lahko vsebuje množico opisov spletnih storitev, potrebujemo tudi informacijo o tem, katera vrata (*port type*) moramo uporabiti za povezavo do ciljne operacije. Z elementom *portType* smo tako definirali vrata za dostop do operacije, za potrebe primera smo jih poimenovali *Prisotnost*. Vsaka spletna storitev ima lahko tudi več operacij (*operation*), kar pomeni, da potrebujemo poseben element za izbiro prave operacije, v našem primeru *dodaj*.

Zadnji sklop opisa pa vsebuje opis komponent uporabniškega vmesnika. Komponente smo zaradi njihovega vsebinskega pomena razvrstili v skupine, s čimer se izognemo vsebinsko nepovezanim uporabniškim vmesnikom. Tako smo z elementom *groupName* definirali skupino *Vnos*, v njej pa komponento za vnos imena, ki smo jo poimenovali *Ime*. Komponenti smo z elementom *fieldType* določili njen tip, in sicer *Text*, saj gre za vnos znakovnega niza. Pridružili smo ji tudi pravila za izvajanje postopka validacije. V našem primeru gre za dve pravili, prvo

pravilo se imenuje *NotNull*, ki zahteva vnos podatka, torej je izpolnjevanje komponente obvezno, medtem ko drugo pravilo z nazivom *NotEmpty* dovoli le vnos vidnih znakov, presledki in ostali kontrolni znaki pa niso dovoljeni.

```
<ft:form xsi:schemaLocation="http://fri-mag.com/FormTypes form.xsd"
    xmlns:cmn="http://fri-mag.com/CommonTypes"
    xmlns:ft="http://fri-mag.com/FormTypes"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ft:formID>Prisotnost</ft:formID>
    <ft:WSendpoint>
        <cmn:address>http://localhost:8282/testWS/ProcessForm</cmn:address>
        <cmn:serviceName>ProcessForm</cmn:serviceName>
        <cmn:portType>ProcessForm</cmn:portType>
        <cmn:operation>processOrder</cmn:operation>
    </ft:WSendpoint>
    <ft:content>
        <ft:fieldGroup>
            <cmn:groupName>Vnos</cmn:groupName>
            <cmn:fieldList>
                <cmn:field>
                    <cmn:fieldName>Ime</cmn:fieldName>
                    <cmn:fieldType>Text</cmn:fieldType>
                    <cmn:fieldValidationList>
                        <cmn:validator>
                            <cmn:type>NotNull</cmn:type>
                        </cmn:validator>
                        <cmn:validator>
                            <cmn:type>NotEmpty</cmn:type>
                        </cmn:validator>
                    </cmn:fieldValidationList>
                </cmn:field>
            </cmn:fieldList>
        </ft:fieldGroup>
    </ft:content>
</ft:form>
```

Izsek kode 3.1: Primer uporabe WUIIDL.

Tako smo pripravili opis uporabniškega vmesnika za zajem podatkov, ki predstavlja del vhodnih podatkov za izvedbo integracije. Model predvideva, da je opis uporabniškega vmesnika dostopen s pomočjo strežnika, na katerem teče glavni del aplikacije, torej ga skrbniki sistema lahko kadarkoli tudi prilagodijo. Dostop je odvisen od lokacije oz. naslova, ki pa mora biti znan in se ne sme spreminjati.

3.8 Usmeritve za nadaljnji razvoj modela

Definiran model za integracijo spletnih uporabniških vmesnikov na strani strežnika se lahko uporabi kot osnova in se sproti nadgrajuje za potrebe različnih scenarijev. Nekaj možnosti za izboljšanje modela smo identificirali že med razvojem prototipa, in sicer izboljšavo metode za lažje določanje vsebinske informacije komponent na uporabniškem vmesniku. To bi lahko dosegli s širšim spektrom informacij o posamezni komponenti, s pomočjo katerih bi lahko

natančneje določili kontekst, v katerem je informacija. S tem bi pridobili boljše informacije o vsebini komponente, kar je ključnega pomena za njeno identifikacijo. Prav tako bi želeli izboljšati tudi metodo za uparjanje komponent glede na vsebinsko informacijo, ki ji je namenjena. Možnosti izboljšave modela se kažejo tudi v fazi validacije informacij, za katero bi lahko natančneje definirali prioritete in način preverjanja. Pridemo lahko namreč do stanja, ko za validacijo enake informacije obstajata dve pravili, ki sta si lahko vsebinsko nasprotujoči. Katero pravilo ima pri tem višjo prioriteto? Zasnovani model omenjeno težavo rešuje z enostavnim pravilom, in sicer tako, da upošteva prvo pravilo. Pri definiranju modela smo se osredotočili predvsem na koncept, s katerim smo pokrili osnovni model za integracijo spletnih uporabniških vmesnikov.

Posamezne sklope modela je možno uporabiti samostojno ali pa model v celoti. Tako lahko nekomu povzroča težave izvedba integracije, nekdo pa opazi prednost uporabe jezika za opis integracijskih točk uporabniških vmesnikov, ki predstavlja glavni prispevek magistrskega dela. Model lahko služi le za oblikovanje rešitve, nekomu pa bo nudil osnovo za nadaljnji razvoj svoje rešitve. V primeru delitve modela na pomembnejše sklope lahko opazimo generičnost modela, kar pomeni, da je vsak sklop uporaben, in sicer na različnih problemskih domenah.

4 Implementacija arhitekturnega modela

Pri pripravi prototipa modela smo se osredotočili na podporo integracije spletnih uporabniških obrazcev, ki so sestavni del poslovnih aplikacij. Tako smo na primeru implementacije modela pokazali, na kakšen način je možna uporaba modela in kako so razdeljene naloge med posameznimi deli modela. Za slednje smo poskušali izbrati čim bolj primerne tehnologije, ki so trenutno na voljo in dovoljujejo brezplačno uporabo.

4.1 Uporabljena orodja in tehnologije

Pri implementaciji prototipa smo najprej izbrali primerne tehnologije, s katerimi smo lahko udejanjili posamezne dele modela za integracijo spletnih uporabniških vmesnikov. Za izhodišče smo pri vsakem izmed ključnih delov modela identificirali nabor tehnologij, ki bi jih lahko uporabili. Ker smo se osredotočili izključno na spletne uporabniške vmesnike, smo pri izbiri tehnologij prav tako ostali v teh okvirih. Pri izbiri smo bili pozorni tudi na modernejše tehnologije, sicer pa smo bili pri izbiri pazljivi, saj nismo želeli uporabiti neuveljavljenih tehnologij.

Pri izboru programskega jezika, s pomočjo katerega smo pripravili implementacijo prototipa, nismo imeli večjih težav. Glede na izkušnje in primernost jezika za tovrstne rešitve smo izbrali visokonivojski programski jezik Java. Zanj smo se odločili tudi zaradi odlične združljivosti z ostalimi tehnologijami, prav tako smo s programskim jezikom neposredno določili tudi platformo, na kateri bo temeljila rešitev. Platforma Java EE je trenutno ena bolj priljubljenih, na njenih temeljih se razvijajo novejši informacijski sistemi. Aktualna različica platforme Java EE ima oznako 1.7 s podporo programskemu jeziku Java z različico 7, medtem ko ima platforma Java SE trenutno na voljo različico z oznako 1.8 in nudi podporo programskemu jeziku Java z različico 8. Z izbiro programskega jezika Java dopuščamo tudi možnost integracije z obstoječim informacijskim sistemom.

Pri izbiri razvojnega okolja smo se osredotočili predvsem na pogoje, možnost brezplačne uporabe, podporo za razvoj s programskim jezikom Java in možnost uporabe v okolju Windows. Tem pogojem sicer ustreza več razvojnih okolij, z vidika boljših uporabniških izkušenj pa smo se na koncu odločili za razvojno okolje Eclipse. Okolje Eclipse temelji na modelu IDE (*Integrated Development Environment*), ki združuje množico orodij tako za pisanje programske kode kot nastavljanje parametrov za zagon aplikacij in njihovo razhroščevanje. Cilj takšnih orodij je, da razvijalcu omogočajo dostop do zbirke modulov, ki jih potrebuje v danem trenutku.

Razvili smo prototip modula za integracijo spletnih uporabniških vmesnikov, zato smo za njegovo testiranje potrebovali tudi ustrezen aplikacijski strežnik, na katerega smo namestili osrednjo aplikacijo. Za strežniške aplikacije, ki temeljijo na platformi Java EE, je na voljo večje

število strežnikov, med bolj priljubljenimi so Apache Tomcat, Jetty, Glassfish, JBoss in IBM WebSphere. Če gledamo z vidika potreb za razvoj prototipa, med njimi ni bistvenih razlik, v primeru produkcijske različice strežniške aplikacije pa je treba upoštevati dodatne parametre, na primer optimalno združljivost s strojno opremo. Zaradi predhodne uporabe in osvojenega znanja smo se odločili za strežnik Apache Tomcat. Med drugim je združljiv tudi z razvojnim okoljem Eclipse.

Za boljšo razumljivost in lažje upravljanje programske kode obstajajo temu namenjena programska ogrodja, ki ponujajo tudi vzdrževanje medsebojne odvisnosti med posameznimi moduli ter prav tako lažje izvajanje faze gradnje (*build*) in faze postavitve na strežnik (*deploy*). V ta namen smo izbrali ogrodje Apache Maven, ki prvenstveno služi za upravljanje projekta, s stališča razvijalca pa ima takšno ogrodje dobrodošlo lastnost, saj predpisuje smernice za nadaljnji razvoj, poleg tega vsebuje tudi zadostno količino informacij o projektu. Ogrodje Apache Maven sicer temelji na konceptu modela POM (*Project Object Model*), ki v osnovi pomeni definiranje pravil razvoja na nivoju posameznega modula. S tem dosežemo delitev programske kode, ki tako postane bolj razumljiva in lažja za vzdrževanje. Vse informacije so zapisane v datoteki POM, ki temelji na označevalnem jeziku XML, s predpisano strukturo. Pri definiranju postopkov za gradnjo in postavitev projekta si razvijalec lahko pomaga tudi z vtičniki (*plugins*), s pomočjo katerih lahko izvede določene procese, kot je kopiranje datotek v ciljno mapo. Vtičniki so lahko pripravljeni v tretjih organizacijah, lahko pa jih razvijalec pripravi v lastni izvedbi. Uporaba tovrstnih ogrodij postaja ustaljena praksa, saj očitno pripomorejo k boljšemu razumevanju in upravljanju projekta.

Za izmenjavo podatkov med različnimi tehnologijami potrebujemo opisni jezik, s katerim bomo lahko natančno določili obliko, strukturo in način prenosa podatkov. Za potrebe izmenjave podatkov sicer obstaja več opisnih jezikov. Najbolj znan je označevalni jezik XML, sledi pa mu oblika zapisa JSON (*JavaScript Object Notation*), ki se je razvila predvsem za lažje notiranje objektov v skriptnem jeziku JavaScript, po katerem je dobila tudi naziv. Zaradi boljše združljivosti s programskim jezikom Java smo se odločili za opisni jezik XML, ki ga bomo uporabili za strukturiranje podatkov med njihovim prenosom. Za definicijo strukture XML potrebujemo shemo, s katero predpišemo določena pravila, ki jih je treba upoštevati pri končni strukturi dokumenta XML. Pravila opišemo v obliki sheme, trenutno pa je najbolj razširjen opisni jezik XSD. Z njim lahko natančno definiramo elemente podstruktur, ki na koncu tvorijo celotno strukturo dokumenta XML. Prednost omenjene tehnologijo je v tem, da je uveljavljena in združljiva s preostalimi ogrodji, ki smo jih že izbrali, hkrati pa na podlagi sheme izvajamo tudi osnovni proces validacije dokumenta XML, s katero preverimo, ali je dokument veljaven, glede na pravila, definirana v shemi.

Odločiti smo se morali tudi za način, s katerim bomo izvedli dinamično gradnjo spletnega uporabniškega vmesnika. To je faza integracije, kjer moramo na podlagi vhodnih podatkov

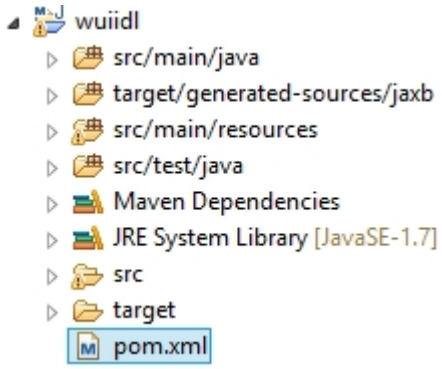
pripraviti končni spletni uporabniški vmesnik, ki bo dostopen uporabniku. Pri gradnji spletnega uporabniškega vmesnika smo morali upoštevati tudi spremljanje dogodkov na uporabniškem vmesniku in ravnanje s spremembami, ki so odraz dogodkov. Prav tako smo morali podpreti postopek validacije na podlagi vnaprej določenih pravil, ki smo jih aplicirali na komponente uporabniškega vmesnika. Izbirali smo med množico znanih ogrodij, kot so Spring MVC (*Spring Model - View - Controller*), JSF, Vaadin, GWT in Apache Wicket. Zaradi agilnega pristopa, ki ga omogoča, smo izbrali prav slednjega, torej Apache Wicket. Izbrano ogrodje ima podporo za predloge, ki temeljijo na značkah označevalnega jezika HTML, in sicer z dodatnimi notacijami, razumljivimi le ogrodju Apache Wicket. Prav omenjene notacije omogočajo dinamičnost priprave končne podobe spletnega uporabniškega vmesnika.

Za podporo določenih aktivnosti modela smo morali vključiti tudi arhitekturo spletnih storitev. Za njihovo implementacijo smo morali izbrati ustrezne tehnologije. Potrebovali smo namreč metodo za označitev spletne storitve in tehnologijo za podporo komunikacije. Za potrebe opisovanja stičnih točk spletne storitve smo izbrali opisni jezik WSDL, ki velja za eno izmed uveljavljenih metod. Opis spletne storitve vključuje tudi sheme za opis struktur podatkov, ki bodo poslani v zahtevku ali odgovoru spletne storitve. V opisu je treba določiti tudi mehanizem, na katerem bo temeljila komunikacija do spletne storitve, izbiramo lahko med arhitekturnim stilom REST in protokolom SOAP (*Simple Object Access Protocol*). Glavna razlika med njima je v tem, da protokol SOAP omogoča definiranje strukture podatkov, tako na strani odjemalca kot tudi na strani storitve [26]. Oba protokola za prenos podatkov uporabljata protokol HTTP (*Hypertext Transfer Protocol*), le da protokol SOAP uporablja izključno metodo POST. Protokol SOAP na področju varnosti dopušča več možnost (*WS-Security*) kot le HTTPS (*HTTP Secure*) in SSL (*Secure Sockets Layer*). Omogoča tudi podpis sporočila, s čimer zagotovimo njegovo celovitost med prenosom, prav tako pa sporočilo lahko tudi šifriramo, kar zagotavlja zaupnost med pošiljateljem in prejemnikom. Glede na to, da smo želeli vnaprej definirati strukturo sporočil za komunikacijo med spletno storitvijo in osrednjo aplikacijo, smo se odločili za uporabo protokola SOAP. Njegova sporočila temeljijo na ovojnici (*envelope*), ki je sestavljena iz zaglavja (*header*) in vsebinskega dela (*body*), ovojnica pa ne definira strukture podatkov znotraj vsebinskega dela. Tega namreč obravnava kot sporočilo, ki ga je treba prenesti s točke A do točke B, pri tem pa zagotoviti da se vsebina med prenosom ni spremenila.

4.2 Struktura programskih modulov

Postavitve strukture projekta v razvojnem okolju Eclipse smo začeli z razčlenitvijo modela na module, s pomočjo katerih smo lahko glede na vsebino oz. pomen ločili programsko kodo in ostale vire. Znano je, da nam moduli na takšen način omogočajo upravljanje različic in ostalih sprememb, ki jih želimo obvladovati, prav tako jih med seboj po potrebi lahko vključujemo oz.

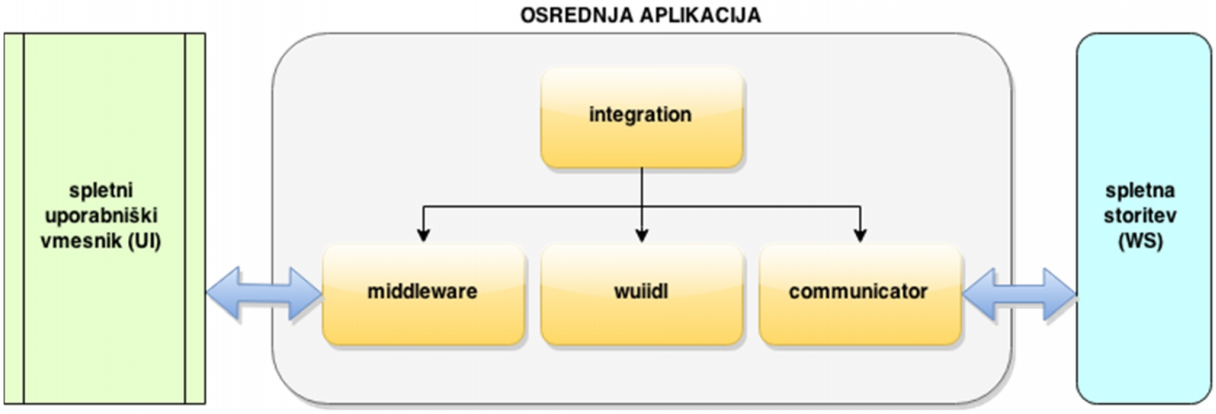
izključujemo. S tem pridobimo preglednost programske kode. Skratka aplikacija je boljše strukturirana (Slika 4.1) in razdeljena na več delov, ki jih v tem primeru imenujemo moduli.



Slika 4.1: Primer strukture modula *wuiidl*.

Glede na to, da smo za delitev po modulih izbrali tehnologijo Apache Maven, smo z njo pridobili še dodatne prednosti modela POM (*Project Object Model*), ki smo ga omenili že v poglavju 4.1 z opisom izbire tehnologij. Model POM je v praksi predstavljen z eno samo datoteko, imenovano *pom.xml*, ki je običajno v korenski mapi modula (Slika 4.1). Datoteka je v osnovi razdeljena na več sklopov, in sicer sklop s splošnimi informacijami modula, sklop z navodili in nastavitvami za gradnjo, sklop z informacijami o projektu in sklop z nastavitvami razvojnega okolja. Sklop s splošnimi informacijami vsebuje osnovne informacije o modulu, ki zajemajo različico modula, naziv modula in pripadajočega artefakta, vključno z informacijami o relaciji modula do preostalih modulov. Pri tem ne smemo pozabiti, da se lahko sklicujemo tudi na javno objavljene module, ki so pripravljeni za uporabo v izbrane namene. Ti moduli so običajno objavljeni v javnih shrambah, do katerih lahko dostopamo. Sklop z navodili in nastavitvami za gradnjo modula opisuje predvsem postopke ter parametre za gradnjo modula. Sklop z informacijami o projektu vsebuje vrste licenc, pod katerimi se projekt izdaja, naziv organizacije, ki zastopa projekt, seznam razvijalcev, ki so prispevali k razvoju projekta in podatke o vlagateljih, ki so finančno ali materialno podprli projekt. Nastavitve razvojnega okolja, ki so skupne vsem razvijalcem in jih redko prilagajamo, so zabeležene v ločenem sklopu informacij, ki predstavljajo vsebino datoteke POM ter imajo v ogrodju Apache Maven glavno vlogo.

Pred razvojem prototipa smo projekt vsebinsko razdelili na več manjših modulov, ki smo jih poimenovali *integration*, *wuiidl*, *middleware* in *communicator* (Slika 4.2), vsi skupaj pa predstavljajo prototip modela. Module smo strukturirali hierarhično, in sicer smo za korenski oz. glavni modul definirali modul z nazivom *integration*, ki vsebuje tudi relacije na svoje podrejene module. Med njimi je modul z nazivom *wuiidl*, pripravili smo ga izključno z namenom definiranja razredov, ki temeljijo na shemah XSD.



Slika 4.2: Struktura modulov osrednje aplikacije.

Modul z glavno vlogo smo poimenovali *middleware*, vsebuje pa programsko kodo, s katero smo poskrbeli tako za odločanje kot tudi za pripravo integriranih spletnih uporabniških vmesnikov. Ob koncu smo definirali še modul z nazivom *communicator*, ki ima vlogo posrednika med modelom za izvedbo integracije in spletnimi storitvami. V naslednjih odstavkih bomo opisali njihove glavne naloge.

4.3 Programski modul *integration*

Modul ima vlogo zbirnega modula in je bil ustvarjen izključno z namenom vzpostavitve relacij na zunanje odprtokodne knjižnice v obliki artefaktov, ki smo jih potrebovali za izvedbo prototipa. V tem primeru modul *integration* predstavlja starša preostalih modulov (Izsek kode 4.1), ki so hierarhično gledano takoj pod njim. Na takšen način zagotavlja posreden dostop do preostalih modulov in tako zagotovimo centraliziran dostop do skupne programske kode.

```
<groupId>com.fri.mag</groupId>
<artifactId>integration</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>pom</packaging>

<modules>
    <module>communicator</module>
    <module>wuiidl</module>
</modules>

<properties>
    <log4j.version>1.2.17</log4j.version>
    <slf4j.version>1.7.5</slf4j.version>
    <junit.version>4.11</junit.version>
    <jersey.version>1.18.1</jersey.version>
</properties>
```

Izsek kode 4.1: Izsek datoteke POM, centralni nadzor različic.

S korenskim modulom običajno rešujemo problem modulov, za katere želimo, da jih upravljamo centralizirano. Tako se oblikuje modul, s katerim lahko nadziramo različice modulov in ostalih artefaktov (Izsek kode 4.1), ki jih uporabljamo. Vlogo nadzornega modula ima v našem primeru modul *integration*, ki ne vsebuje programske kode, kar pa je zgolj naključje.

4.4 Programski modul *wuiidl*

Glavna vloga modula *wuiidl* je generiranje razredov na podlagi definiranih shem XSD, ki definirajo strukturo jezika za opisovanje stičnih točk integracije. Razrede generiramo s pomočjo orodja XJC (*JAX Binding Compiler*) (Izsek kode 4.2), ki je v osnovi namenjen prav pretvarjanju shem XSD v pripadajoče razrede. Modul vsebuje sheme XSD, iz katerih mora na podlagi pravil v datoteki POM generirati razrede Java. Z vsakim zagonom gradnje modula se zažene generiranje razredov in tako prilagodimo njihovo vsebino. Razrede uporabljajo tudi preostali moduli, nanje pa se sklicujejo neposredno.

```
<plugins>
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jaxb2-maven-plugin</artifactId>
        <version>1.6</version>
        <executions>
            <execution>
                <id>xjc</id>
                <goals>
                    <goal>xjc</goal>
                </goals>
            </execution>
        </executions>
        <configuration>
            <target>2.1</target>
            <noGeneratedHeaderComments>true</noGeneratedHeaderComments>
            <extension>true</extension>
            <schemaDirectory>
          ${project.basedir}/src/main/resources/xsd
        </schemaDirectory>
        <bindingDirectory>
          ${project.basedir}/src/main/resources/xsd
        </bindingDirectory>
            <outputDirectory>
          ${project.build.directory}/generated-sources/jaxb
        </outputDirectory>
            <packageName>com.fri.mag.jaxb.generated</packageName>
        </configuration>
    </plugin>
</plugins>
```

Izsek kode 4.2: Izsek datoteke POM, konfiguracija vtičnika XJC.

Zaradi preprečitve kasnejšega spreminjanja generiranih razredov smo za ciljno lokacijo razredov določili mapo z nazivom *target*. Slednja vsebuje implicitno pravilo, da so v njej že prevedene datoteke razredov, s čimer smo razvijalcu onemogočili kasnejše spreminjanje generiranih razredov. Za boljšo preglednost programske kode smo definirali tudi pakete, s katerimi programsko kodo razdelimo na manjše skupine glede na obravnavano vsebino.

4.5 Programski modul *middleware*

Glavni modul smo poimenovali *middleware*. Vsebuje namreč kodo, ki vsebuje logiko za izvajanje integracije, postopke validacije in tudi procesiranje podatkov. Poleg tega skrbi za pripravo spletnega uporabniškega vmesnika, ki predstavlja rezultat prototipa. Ta programski modul ima tudi vlogo končne aplikacije, ki jo naložimo na aplikacijski strežnik. Modul ima tako ključno vlogo pri izvedbi prototipa modela za integracijo spletnih uporabniških vmesnikov. Omenjeni programski modul vzdržuje povezavo do nekaterih pomembnejših knjižnic, s katerimi smo si pomagali pri izvedbi integriranega spletnega uporabniškega vmesnika. Takšni knjižnici sta v našem primeru *wicket-core* in *wicket-extensions*, ki smo ju vključili z namenom uporabe ogrodja Apache Wicket. Na podoben način smo vključili tudi preostale module, ki jih potrebujemo za izvajanje prototipa na aplikacijskem strežniku.

4.5.1 Uparjanje opisov komponent

Pri razvoju prototipa modela za integracijo spletnih uporabniških vmesnikov smo ugotovili tudi nekatere težave, kot je uparjanje opisov komponent oz. v našem primeru vnosnih polj. Težavo predstavlja ugotavljanje vrste obravnavane informacije za izbrano komponento, kakšna je njena vsebina in v kakšnem kontekstu se uporablja. Poleg tega lahko pride tudi do različnih oblik vnosa za vsebinsko enake opise, torej se moramo vprašati, katero obliko vnosa izbrati oz. katera oblika ima višjo prioriteto. Postopek uparjanja vnosnih polj, ki smo jih uporabili v prototipu, je lahko zelo kompleksen in hkrati tudi problematičen. V postopku je treba ohranjati informacije o izvornih in končnih opisih vnosnih polj, kar pomeni, da je treba hraniti preslikave, ki smo jih naredili v postopku uparjanja. Preslikave nam bodo namreč ob koncu procesiranja podatkov z vmesnika služile predvsem za identifikacijo posameznega polja na izbranem uporabniškem vmesniku, s čimer si bomo lahko izoblikovali zahtevek za procesiranje podatkov posameznega uporabniškega vmesnika.

V magistrskem delu smo se osredotočili predvsem na razvoj modela za integracijo spletnih uporabniških vmesnikov in se zato nismo posebej posvečali podrobnostim. Ena od teh je razvoj posebne metode, s katero bi za posamezno vnosno polje natančneje določili vsebinski pomen podatka s kontekstom, v katerem je ta, in tako ugotovili, kakšna oblika vnosa bi bila zanj najbolj primerna. Za potrebe prototipa smo tako izbrali najosnovnejši pristop, s katerim smo lahko pokrili večino zahtev, ki smo jih imeli ob razvoju. Za uparjanje opisov vnosnih polj smo izbrali pristop uparjanja po nazivu vnosnega polja (Izsek kode 4.3). Za vsak opis vnosnega polja, ki

smo ga zaznali na posameznem vmesniku, smo najprej pridobili pripadajoči naziv, ki smo ga kasneje določili za ključ opisa. Če smo našli naslednji opis komponente z enakim nazivom, smo ga pridružili obstoječemu, le da pridružitev v tem primeru ni pomenila nič drugega kot to, da smo za trenuten opis pregledali pravila validacije in jih v primeru neobstoja dodali na obstoječo listo pravil. Slabost omenjene rešitve je v tem, da smo morali v času razvoja predpostavljati, da bodo za potrebe prototipa opisi vmesnikov ustrezno prilagojeni. Vsebinsko enake komponente bodo torej imele tudi enak naziv. S tem smo poenostavili postopek za uparjanje komponent, rešitev pa kljub temu ustreza osnovnim zahtevam prototipa.

```
public void mergeSelectedForms() {
    IntegrationFieldGroup group = null;
    for(String formID : selectedForms) {
        FormType form =
                FormDefinitionManager.getInstance().getFormById(formID);
        // skupine podatkov za trenutni uporabniški vmesnik
        List<FieldGroupType> fieldGroups =
                form.getContent().getFieldGroups();
        for(FieldGroupType fieldGroup : fieldGroups) {
            String groupName = fieldGroup.getGroupName();
            // preveri, če skupina že obstaja ...
            group = this.fieldGroupMap.get(groupName);
            // sicer jo ustvari in dodaj
            if(group == null) {
                group = new IntegrationFieldGroup(groupName);
                this.fieldGroupMap.put(groupName, group);
            }
            // v primeru, da ne obstajajo, dodaj podatke
            for(FieldType field : fieldGroup.getFieldList().getFields()) {
                if( !group.getFieldList().containsField(field) ) {
                    group.getFieldList().add(new IntegrationField(field));
                }
            }
        }
    }
}
```

Izsek kode 4.3: Izsek implementacije za uparjanje komponent.

Ob tem smo zaznali še eno težavo, ki smo jo že omenjali, to je, kako zagotoviti, da bodo vsebinsko enaka vnosna polja imela tudi enako obliko vnosa, torej, da ne bo prihajalo do razkola pri obliki vnosnega polja. Zato smo za potrebe prototipa predpostavili, da bodo opisi vmesnikov prilagojeni, in sicer na način, da bodo na nivoju vnosnih polj skladni s pravili, ki smo jih pravkar omenili. Vsebinsko enaka vnosna polja bodo torej imela enake nazive in prav tako enako obliko vnosa. S tem smo poenostavili rešitev problema, predlog boljše rešitve je implementacija mehanizma za iskanje oblike vnosa, ki bo skupna vsem navzočim vnosnim poljem.

Pravkar smo identificirali določene omejitve prototipa, ki v tem primeru ne predstavljajo večjih ovir, za boljšo rešitev problemov pa bi, kot smo že omenili, potrebovali posebno

obravnavo problema, s pomočjo katere bi definirali ustrezno metodologijo za doseganje želenega cilja.

4.5.2 Prikaz komponent na spletnem uporabniškem vmesniku

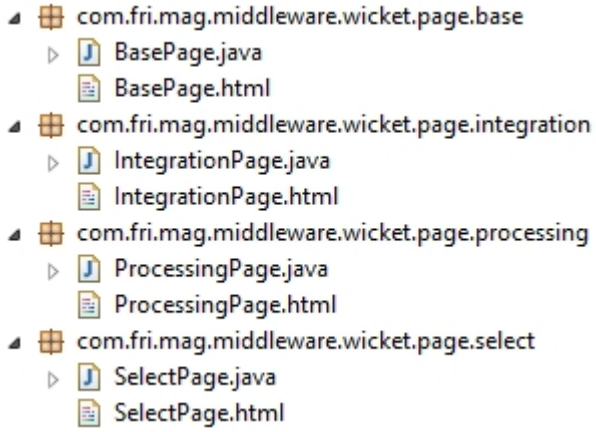
Najpomembnejši del integracije je njen končni rezultat, ki je v našem primeru viden uporabniku. Govorimo o spletnem uporabniškem vmesniku, prek katerega poteka glavna interakcija med uporabnikom in aplikacijo. Zagotoviti smo morali čim bolj optimalen proces prikazovanja komponent na spletnem uporabniškem vmesniku. Pri tem smo imeli več izzivov, in sicer smo se spraševali, kako implementirati prikazovanje komponent vsaj toliko generično, da bi bilo kasnejše dodajanje komponent čim bolj poenostavljeno. Poleg tega smo želeli zagotoviti tudi apliciranje pravil za validacijo posamezne komponente, v našem primeru vnosnega polja. Pojavil se je tudi dodaten pomislek glede združevanja komponent v skupine, ki uporabniku pomagajo pri razumevanju spletnega uporabniškega vmesnika. V skupini naj bi bile komponente, ki jih združuje skupna lastnost podatkov, kot so podatki o naslovu, ki je običajno sestavljen iz več vnosnih polj. Za implementacijo spletnega vmesnika smo uporabili ogrodje Apache Wicket, s katerim nam je v relativno kratkem času uspelo vzpostaviti osnovo spletnega uporabniškega vmesnika.

Posebnost omenjenega ogrodja je v tem, da za vsako izmed komponent, ki je na koncu prikazana na uporabniškem vmesniku, vsebuje tudi pripadajočo predlogo, na katero se sklicuje ob njenem upodabljanju. Predloge vsebujejo glavno informacijo o strukturi spletnega uporabniškega vmesnika, torej hierarhijo komponent, ki jih vmesnik vsebuje. Poleg tega vsebuje posebne notacije, prek katerih se ogrodje sklicuje na posamezne komponente, in jih prilagaja glede na pripadajočo programsko kodo. Za nekatera dejanja imamo tako na voljo statično implementacijo s pomočjo predlog, v našem primeru predloga predstavlja le osnovni videz spletnega uporabniškega vmesnika. Za preostali del rešitve smo potrebovali veliko več dinamike, ki pa smo jo udejanjili s pomočjo pripadajoče programske kode. Predloga in pripadajoča programska koda sta povezani z imenom datoteke, kar pomeni, da za javanski razred *Form.java* potrebujemo tudi predlogo *Form.html*. Datoteki sta običajno v istem paketu, s čimer zagotavljamo omenjeno pripadnost in boljši pregled kode.

4.5.2.1 Postavitev osnovnih strani

Ogrodje Apache Wicket ima nekaj osnovnih modelov, s katerimi pokriva določene dele spletnih uporabniških vmesnikov. Najosnovnejši model je spletna stran (*WebPage*), ki ima vlogo priprave spletne strani kot celote, torej določen videz, vključno s pripadajočimi meniji, zaglavjem in vznožjem spletne strani. Pripravili smo osnovno stran, s katero smo definirali privzeti videz statičnih odsekov, kot sta zaglavje in vznožje, vsebino pa bomo prilagajali z razširitvami osnovne strani. Na osnovno stran smo vključili tudi datoteke za stilsko oblikovanje spletne strani in določili njen naslov. Glede na potrebe prototipa in demonstracije modela smo se odločili, da pripravimo tri glavne videze strani (Slika 4.3). Prva je stran, namenjena izbiri

spletnih vmesnikov, ki so podani z opisnimi datotekami in jih uporabnik želi vključiti v integracijo. Po uspešni izvedbi integracije se bo na naslednji strani pojavil rezultat modela, torej integriran spletni uporabniški vmesnik. Ob kliku na gumb za procesiranje bo uporabnik v ločenem videzu strani prejel rezultate oddaje podatkov, če bodo vneseni podatki seveda ustrezali vsakemu izmed pravil validacije. S tem smo določili osnovni potek dogodkov na spletnem uporabniškem vmesniku prototipa.



Slika 4.3: Struktura osnovnih strani.

Začetna spletna stran vsebuje seznam opisanih spletnih vmesnikov, ki so na voljo za izvedbo integracije. Tako smo morali najprej razčleniti vse opise, jih prestrukturirati v prototipu poznane objekte in njihove nazive prikazati na seznamu spletnega obrazca. Ob kliku na gumb za oddajo podatkov se uporabniku prikaže integrirani spletni uporabniški vmesnik, vključno s komponentami, ki so bile definirane v izbranih opisih uporabniških vmesnikov. Postopek integracije bomo opisali nekoliko kasneje, zato želimo omeniti le še zaključni del, in sicer prikaz rezultatov procesiranja podatkov, s pomočjo katerih bo uporabnik lahko preveril uspešnost oddaje podatkov. Na zadnji strani smo v primeru uspešne oddaje podatkov za vsakega izmed izbranih vmesnikov izpisali »*OK!*«, če je pri katerem izmed njih med oddajo podatkov prišlo do napake, pa smo izpisali ustrezen opis napake. Uporabnik tako hitro opazi, za kateri uporabniški vmesnik je pravilno izpolnil vnosna polja in pri katerih je storil napako.

4.5.2.2 Prikaz vnosnih polj in njihovih skupin

Omenili smo že, da smo med postopkom integracije oz. priprave spletnega uporabniškega vmesnika kot rezultat integracije vključili tudi združevanje vnosnih polj glede na njihove skupne lastnosti. S tem smo uporabniku približali spletni uporabniški vmesnik v smislu boljšega razumevanja in posledično manjše zmede pri vnosu podatkov. Glede na to, da smo v opisnem jeziku WUIIDL predvideli posebno strukturo za opis skupin, s pomočjo katere lahko določimo skupino skupaj s pripadajočimi vnosnimi polji, smo to lastnost tudi izkoristili. Težava nastane v primeru, ko je vnosno polje z enako oz. podobno vsebino v dveh različnih skupinah, ki pa jih med seboj ločujemo s posebnim identifikacijskim nazivom. Težavi smo se izognili s tem, da

smo upoštevali prvo pojavitev opisa vnosnega polja in ponovne pojavitve vnosnega polja z enakim identifikacijskim nazivom nismo posebej obravnavali. Za potrebe prototipa tovrstna rešitev sicer zadošča, v primeru projektne uporabe pa bi bilo treba definirati dodatna pravila in metode, s katerimi bi ustrezno rešili takšna stanja.

Prikazovanje skupin komponent smo želeli implementirati čim bolj generično, zato bi morali skupine prikazati neodvisno od oblike vnosnega polja, ki jo zahteva njegov opis. Če smo želeli uresničiti omenjeno vizijo generičnosti, smo morali posamezna vnosna polja in pripadajoče oznake, ki uporabniku služijo za identifikacijo vnosnega polja, implementirati kot eno samo komponento (Izsek kode 4.4), ki pa smo jo nato pridružili skupini preostalih, po vsebini podobnih komponent (*FieldGroupPanel*).

```
private FieldPanel buildFieldPanelForField(String panelId,
            ListItem<IntegrationField> item) {
    FieldTypeEnum fieldType = FieldTypeEnum.fromValue(
                                    item.getModelObject().getFieldType() );
    switch(fieldType) {
        case TEXT:
            return new TextFieldPanel(
                    panelId,
                    new Model<IntegrationField>(item.getModelObject()) );
        case DROP_DOWN:
            return new DropDownFieldPanel(
                    panelId,
                    new Model<IntegrationField>(item.getModelObject()) );
        case NUMBER:
            return new NumberFieldPanel(
                    panelId,
                    new Model<IntegrationField>(item.getModelObject()) );
        case RADIO:
            return new RadioFieldPanel(
                    panelId,
                    new Model<IntegrationField>(item.getModelObject()) );
        case CHECK_BOX:
            return new CheckBoxFieldPanel(
                    panelId,
                    new Model<IntegrationField>(item.getModelObject()) );
        default:
            System.err.println("Field type is not supported!");
            return null;
    }
}
```

Izsek kode 4.4: Izsek implementacije za prikaz komponent.

Za pripravo komponente smo uporabili plošče (*panel*), na katerih smo združili elemente, ki skupaj tvorijo opisu pripadajočo komponento (*FieldPanel*). S tem smo pripravili strukturo, ki jo lahko kadarkoli nadgradimo z dodatnimi oblikami vnosnih polj.

Nekaj težav smo imeli tudi pri dinamičnem prikazovanju komponent, bodisi skupin ali posameznih vnosnih polj, saj je s predlogami težje zagotoviti takšno stopnjo dinamike, poleg

tega pa obdržati tudi generičnost, ki smo si jo postavili za cilj. Za tovrstno prikazovanje smo uporabili vnaprej pripravljen koncept ogrodja Apache Wicket, in sicer model *ListView* (Izsek kode 4.5), ki je prav tako pripravljen generično, vendar smo pri tem morali biti previdni. V tem primeru imajo glavno vlogo modeli, ki so v ogrodju Apache Wicket namenjeni prenašanju in hranjenju informacij o podatkih, ki jih upravlja spletni uporabniški vmesnik. Od njih je odvisno, kako bodo potekali procesi validacije in kakšni podatki bodo prispeli do postopka za oddajo podatkov. Omenjeni koncept z modelom *ListView* vsebuje posebnost, in sicer sam poskrbi za apliciranje vseh potrebnih parametrov, le prikaz komponente je prepuščen razvijalcu. To nam je omogočilo prikaz podobnih komponent znotraj iste skupine, in sicer kot listo komponent oz. vnosnih polj.

```
private void initComponents() {
    ListView<IntegrationField> fieldList = new ListView<IntegrationField>(
        "fieldList",
        getModelObject().getFieldList() ) {
            @Override
            protected void populateItem(ListItem<IntegrationField> item) {
                final FieldPanel panel =
                    FieldGroupPanel.this.buildFieldPanelForField(
                        "field",
                        Item );
                item.add(panel);
            }
        };
    fieldList.setReuseItems(true);
    this.add(fieldList);
    this.add(new Label("groupCaption",
                       Model.of(getModelObject().getGroupName())));
}
```

Izsek kode 4.5: Izsek implementacije za dinamični prikaz komponent.

Za prikaz vnosnega polja kot takšnega smo uporabili že s strani ogrodja Apache Wicket pripravljene koncepte, s čimer smo skrajšali razvoj prototipa in bili pri tem dovolj agilni. Kot primer prikaza vnosnega polja za vnos znakovnega niza (Izsek kode 4.6) smo namreč uporabili koncept modela *TextField*, ki je prav tako pripravljen generično, torej mu lahko določimo tip vrednosti, ki naj bi jo vsebovalo vnosno polje.

```
    @Override
    protected void initComponents() {
        TextField<String> field = new TextField<String>(
                "fieldValue",
                new PropertyModel<String>(getModel(), "fieldValue"));
        field.setLabel(Model.of(getModelObject().getFieldName() + ":"));
        this.applyValidators(field);
        add(field);
    }
```

Izsek kode 4.6: Izsek implementacije za prikaz komponente.

Pri procesiranju podatkov s spletnega uporabniškega vmesnika nam tako ni bilo treba skrbeti za vnosno polje in pripadajoče dogodke, ki se morajo zgoditi pred ali po procesiranju podatkov. Na posamezno vnosno polje smo aplicirali tudi zahtevana pravila za izvajanje postopkov validacije, ki se izvedejo takoj po kliku na gumb za oddajo podatkov. S tem preprečimo kasnejše težave pri procesiranju podatkov.

4.5.3 Implementacija pravil za izvajanje validacije podatkov

Izvajanje validacije je pomembnejši korak pred procesiranjem podatkov, vnesenih v spletni uporabniški vmesnik. S tem lahko uporabniku že pred poskusom procesiranja podatkov sporočimo, pri katerih vnosnih poljih je storil napako in na kakšen način jo lahko odpravi oz. kakšna oblika podatka je pričakovana v določenem vnosnem polju. V primeru napak se namreč procesiranje podatkov ne izvede in se uporabniku prikaže spletni uporabniški vmesnik z opisom vseh napak, ki so bile identificirane na oddanih podatkih.

V našem primeru smo za potrebe prototipa implementirali nekatera najbolj pogosta pravila za izvajanje validacije in jih vključili v integracijo. Vgradnjo pravil smo implementirali v smislu dodajanja pravil po potrebi, glede na opis vnosnega polja (Izsek kode 4.7), v katerem so podani parametri za njihov opis. Za izvajanje pravil validacije smo uporabili že obstoječe koncepte ogrodja Apache Wicket, kot so *StringValidator*, *PatternValidator*, *EmailAddressValidator* in *DateValidator*. Obstaja tudi možnost implementacije lastnih pravil, ki bi nam koristila v primeru preverjanja neobičajnih omejitev. Prav tako smo tudi v primeru validacije obdržali generičnost koncepta, ki nam je kasneje koristila za enostavnejše dodajanje novih pravil. S tem ohranjamo želeno prilagodljivost uporabe, k čemur stremimo že od začetka razvoja prototipa.

```
private void applyValidators(TextField<String> field) {
    super.applyBasicValidators(field);
    for (IValidator<?> validator : getModelObject().getValidatorList()) {
        if (validator instanceof StringValidator) {
            field.add((StringValidator) validator);
        } else if (validator instanceof PatternValidator) {
            field.add((PatternValidator) validator);
        }
        else {
            System.err.println("Validator '" +
                    validator.getClass().getSimpleName()+
                    "' is not supported for text field!");
        }
    }
}
```

Izsek kode 4.7: Izsek implementacije za nastavljanje pravil.

Pri tem velja omeniti tudi pomanjkljivosti trenutne implementacije, in sicer se moramo zavedati, da smo z omenjeno implementacijo zagotovili le preverjanje podatkov s pomočjo statične validacije, ki je podana skupaj z opisom posameznega vnosnega polja. Pri tem pa ne

smemo pozabiti na možnost dinamične validacije, ki temelji na klicu spletne storitve, ki pošlje zahtevek za preverjanje podatka, in če ta ne ustreza, v odgovoru sporoči opis napake ter namig za njeno odpravo. Poleg že omenjene validacije pa obstaja tudi medsebojna validacija, ki temelji na soodvisnosti med podatki, česar pa ni možno izvesti z infrastrukturo statične validacije, zato bo za takšne primere treba poskrbeti z ustrezno nadgradnjo modela za integracijo spletnih uporabniških vmesnikov.

4.5.4 Procesiranje podatkov

Po uspešni izpolnitvi podatkov v uporabniškem vmesniku mora uporabnik za nadaljevanje klikniti na gumb za oddajo podatkov. S to akcijo uporabnik izkaže željo po zaključku procesa integracije uporabniških vmesnikov, kar v našem primeru pomeni oddajo podatkov na izbrane spletne storitve, ki so definirane z opisi vmesnikov. V tej fazi se mora torej izvesti procesiranje podatkov, ki jih je uporabnik pred tem vnesel v za to pripravljena vnosna polja. Glede na to, da smo v začetni fazi morali združevati opise vmesnikov, moramo pri procesiranju izvesti njihovo ločitev, in sicer glede na opise vmesnikov. Na tem mestu moramo zato dostopati do informacij, ki so podane v opisih vmesnikov. Samo z njihovo pomočjo bomo lahko vnesene podatke ločili v skupine podatkov, ki jih bomo kasneje oddali na točno določene spletne storitve.

V ta namen smo pripravili razred *FormProcessor*, ki nudi podporo ločevanju (Izsek kode 4.8) in pripravi podatkov. Postopek za ločevanje podatkov temelji izključno na opisih vmesnikov, ki so bili vključeni v integracijo. S tem zagotavljamo, da bodo podatki poslani samo na tiste spletne storitve, ki pripadajo izbranim spletnim uporabniškim vmesnikom. Priprava podatkov se začne s pridobivanjem opisa uporabniškega vmesnika, ki pripada prvemu izmed izbranih, nato sledi pregled opisanih komponent za vnos podatkov, nato pa za vsako izmed njih poskušamo najti podatek, ki ji pripada. Pri tem pa ne smemo pozabiti na skupine, ki so definirane v sklopu opisa vmesnika, s katerimi smo omogočili združitev komponent uporabniškega vmesnika v skupine. Po uspešni pripravi vseh vnesenih podatkov za izbrani uporabniški vmesnik smo pripravljeni na njihovo posredovanje spletni storitvi, katere informacije za dostop so prav tako navedene v opisu vmesnika.

```
// preveri vse uporabniške vmesnike
for( String formId : integration.getSelectedForms() ) {
    // poišči definicijo preko formId
    form = FormDefinitionManager.getInstance().getFormById(formId);
    // pripravi listo podatkov za procesiranje
    List<IntegrationField> processingFieldList =
            new ArrayList<IntegrationField>();
    // lista skupin podatkov za trenutni uporabniški vmesnik
    List<FieldGroupType> formFieldGroupList =
            form.getContent().getFieldGroups();
    for(FieldGroupType formFieldGroup : formFieldGroupList) {
        List<FieldType> formFieldList =
                formFieldGroup.getFieldList().getFields();
        // poišči trenutno skupino iz liste skupin
        IntegrationFieldGroup group = integration.getFieldGroupMap().get(
                formFieldGroup.getGroupName());
        // preveri vse podatke v skupini
        for( IntegrationField field : group.getFieldList() ) {
            // poišči podatke iz definicije uporabniškega vmesnika
            if( isFieldDefinedInFormList(formFieldList, field) ) {
                processingFieldList.add(field);
            }
        }
    }
    // oddaja (procesiranje) podatkov
    WSEndpointReferenceType endpoint = form.getWSendpoint();
            result += "\n" + formId + ": " + processForm(
                    createProcessRequest(processingFieldList),
                    endpoint.getOperation(),
                    endpoint.getPortType(),
                    endpoint.getAddress() );
}
```

Izsek kode 4.8: Izsek implementacije za pripravo podatkov na oddajo.

4.6 Programski modul *communicator*

Za potrebe komunikacije s spletnimi storitvami smo razvili poseben modul, ki smo ga poimenovali *communicator*, njegova naloga pa je vzpostavljanje komunikacije z izbrano spletno storitvijo. Modul vsebuje nabor datotek WSDL, v katerih so opisi spletnih storitev in pripadajočih struktur. Datoteke WSDL smo s pomočjo orodja JAX-WS (*Java API for XML Web Services*) uvozili v modul (Izsek kode 4.9), s čimer smo pripravili pripadajoče razrede spletnih storitev. Razredi vsebujejo že pripravljene nastavke za izvedbo komunikacije z določeno spletno storitvijo. S tem imamo pripravljen glavni del implementacije za komuniciranje, pripraviti moramo le še vhodne podatke (zahtevek) in ob koncu izhodne podatke (odgovor) spletne storitve ustrezno obravnavati. Z nastavki spletnih storitev smo vzpostavili komunikacijo z izbrano spletno storitvijo. V modul smo vključili tudi svoj razred z nazivom *Communicator*, ki pa predstavlja vstopno točko modula, torej je namenjen uporabi v preostalih modulih.

```
<plugin>
  <groupId>org.jvnet.jax-ws-commons</groupId>
  <artifactId>jaxws-maven-plugin</artifactId>
  <version>2.2</version>
  <executions>
    <execution>
      <id>wsimport generate-jaxws-sources</id>
      <phase>generate-resources</phase>
      <goals>
        <goal>wsimport</goal>
      </goals>
      <configuration>
        <bindingFiles>
          <bindingFile>common_definitions.xsd</bindingFile>
        </bindingFiles>
        <wsdlFiles>
          <wsdlFile>processForm.wsdl</wsdlFile>
          <wsdlFile>getData.wsdl</wsdlFile>
        </wsdlFiles>
        <sourceDestDir>
          ${project.build.directory}/generated/ws/import
        </sourceDestDir>
      </configuration>
    </execution>
  </executions>
  <configuration>
    <extension>true</extension>
    <target>2.1</target>
      <bindingDirectory>
        ${basedir}/src/main/resources/xsd
      </bindingDirectory>
      <wsdlDirectory>
        ${basedir}/src/main/resources/wsdl
      </wsdlDirectory>
  </configuration>
</plugin>
```

Izsek kode 4.9: Izsek datoteke POM, konfiguracija vtičnika JAX-WS.

4.6.1 Klic spletne storitve *GetData*

Kot enega izmed virov izbirnih vrednosti komponent model predvideva tudi izpostavitev spletne storitve. Zato smo za potrebe prototipa pripravili posebno spletno storitev, ki smo jo opisali v datoteki *'getData.wsdl'*. Vključili smo jo v poseben modul, poimenovan *dataWS*. Za vsako izmed komponent, ki se ločijo po vsebini, smo pripravili svojo operacijo. Pridobivanje seznama poštnih številk smo tako izvedli z uporabo operacije *GetPostNumbers*. Za izvajanje klicev spletnih storitev smo pripravili generično rešitev (Izsek kode 4.10), ki na podlagi vhodnih parametrov pripravi objekte in na koncu izvede klic izbrane operacije spletne storitve.

```
static {
    getDataServiceClient = new GetData();
}

public GetDataResponse getData(GetDataRequest request,
        String operationName, QName portType,
        String WSendpoint) throws Exception {
    GetDataPortType port = getPortType(getDataServiceClient, portType,
            WSendpoint, GetDataPortType.class);
    // pošlji zahtevek na spletno storitev
    GetDataResponse response =
            (GetDataResponse)MethodUtils.invokeExactMethod(
                    port, operationName, request);
    return response;
}
```

Izsek kode 4.10: Izsek implementacije za klic spletne storitve *GetData*.

Implementacija klicev spletnih storitev posredno temelji na datoteki WSDL, na podlagi katere se razredi generirajo. Med njimi je tudi razred *GetData*, ki predstavlja določeno spletno storitev in skrbi za vzpostavitev povezave do spletne storitve. Metoda *getPortType* na podlagi parametrov za izbrano storitev pripravi objekt (Izsek kode 4.11), ki skrbi za izvajanje komunikacije s spletno storitvijo, pred tem pa mu nastavi še njen ciljni naslov.

```
private <T> T getPortType(Service serviceClient, QName portType,
        String WSendpoint, Class<T> clazz) throws Exception {
    synchronized (serviceClient) {
        @SuppressWarnings("unchecked")
        T port = (T)MethodUtils.invokeMethod(serviceClient,
                "get"+portType.getLocalPart());
        setPortProperties((BindingProvider)port, WSendpoint);
        return port;
    }
}
```

Izsek kode 4.11: Izsek implementacije, metoda *getPortType*.

Nastavljanje ciljnega naslova spletne storitve poteka s pomočjo konteksta (Izsek kode 4.12), ki je nosilec informacij, potrebnih za vzpostavitev komunikacije in izvajanje klicev. Med njimi je tudi ciljni naslov spletne storitve, ki smo ga nastavili glede na vrednost, podano v vhodni datoteki, z opisom spletnega uporabniškega vmesnika.

```
private void setPortProperties(BindingProvider bindingProvider,
      String WSendpoint) {
    Map<String, Object> context = bindingProvider.getRequestContext();
    context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, WSendpoint);
}
```

Izsek kode 4.12: Izsek implementacije za ponastavitev ciljnega naslova.

S tem smo opisali uporabo spletne storitve kot vir izbirnih vrednosti izbranih komponent spletnega uporabniškega vmesnika. To je sicer le ena izmed spletnih storitev, uporabljenih v prototipu. V primeru dodatnih bi te morali najprej uvoziti v modul *communicator*, kar omogoča njihovo uporabo.

4.6.2 Klic spletne storitve *ProcessForm*

Za oddajo podatkov, vnesenih s strani uporabnika, potrebujemo opis spletne storitve, ki je namenjena procesiranju podatkov. V primeru prototipa smo spletno storitev za oddajo podatkov opisali v datoteki *'processForm.wsdl'*. Klic spletne storitve je treba izvesti z definicijo zahtevka, ki mora ustrezati opisanim pravilom iz datoteke WSDL. Modul, v katerega smo uvozili spletno storitev za sprejem podatkov, smo poimenovali *testWS*. Zaradi preglednosti smo spletno storitev razdelili na več operacij (Izsek kode 4.13), in sicer glede na uporabniški vmesnik. Obliko sporočil med odjemalcem in spletno storitvijo smo opisali s pripadajočo shemo XSD. Pri tem smo predpostavili, da za oddajo podatkov lahko uporabimo univerzalno shemo. Shemo smo uvozili v datoteko WSDL, na podlagi katere smo pripravili implementacijo spletne storitve. Namen spletne storitve je bil zgolj beleženje zahtevkov in prikaz njihove zgodovine. S tem smo pripravili enostaven pregled prejetih zahtevkov.

```
@Override
@WebMethod(operationName = "ProcessPartner", action =
    "http://com.fri.mag/services/ProcessPartner")
@WebResult(name = "ProcessFormResponse", targetNamespace =
    "http://com.fri.mag/WSType", partName = "ProcessFormResponse")
public ProcessFormResponse processPartner(
        @WebParam(name = "ProcessFormRequest", targetNamespace =
            "http://com.fri.mag/WSType", partName = "ProcessFormRequest")
            ProcessFormRequest processFormRequest) {
    LogStatus.getInstance().appendRequest("processPartner",
            processFormRequest);
    ProcessFormResponse response = new ProcessFormResponse();
    response.setResult("OK");
    response.setStatus(BigInteger.ZERO);
    return response;
}
```

Izsek kode 4.13: Izsek implementacije, operacija *ProcessPartner*.

Podatke izbranega uporabniškega vmesnika s poslanim zahtevkom (Izsek kode 4.14) na ciljni naslov spletne storitve oddamo v obdelavo. Postopek smo ponovili za vse uporabniške vmesnike, ki so bili v prvi fazi integracije vključeni vanjo. Z uspešno oddajo podatkov smo dosegli cilj procesiranja podatkov, uporabniku pa moramo posredovati le še informacije o statusu oddaje. Faza oddaje podatkov ima pomembno vlogo integracije, saj so v primeru njenega neuspeha vsi ostali koraki neuporabni, ker so podatki ostali nedostavljeni.

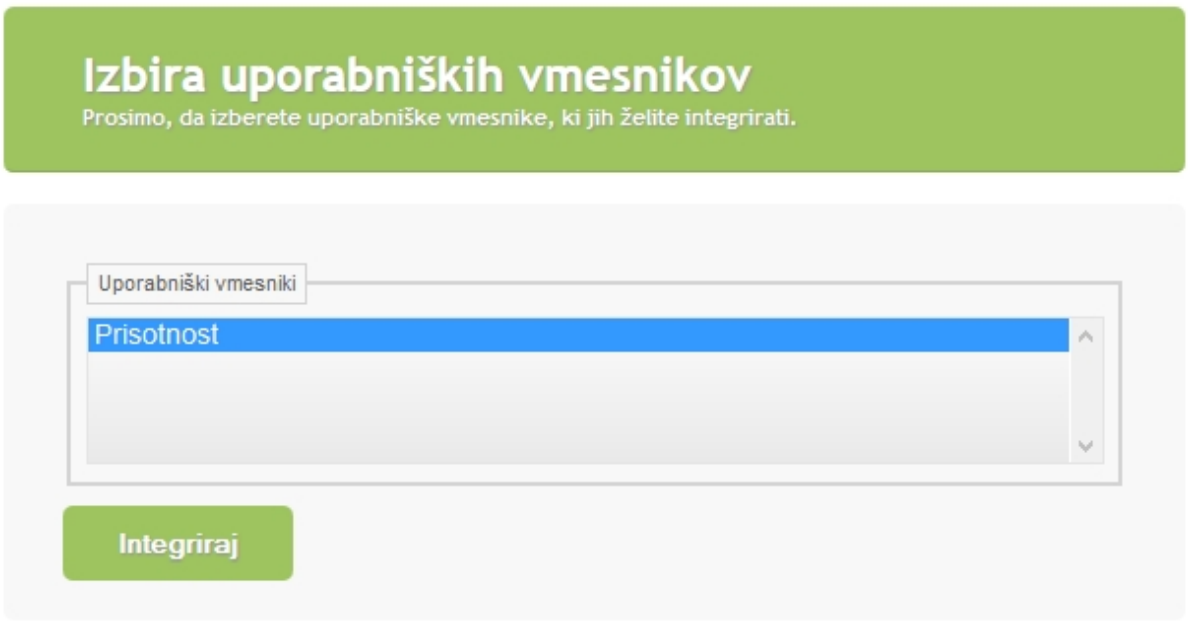
```
public ProcessFormResponse processForm(ProcessFormRequest request,
        String operationName, QName portType,
        String WSendpoint) throws Exception {
    ProcessFormPortType port = getPortType(processFormServiceClient,
            portType, WSendpoint, ProcessFormPortType.class);
    // pošiljanje zahtevka na spletno storitev
    ProcessFormResponse response =
            (ProcessFormResponse)MethodUtils.invokeExactMethod(
                    port, operationName, request);
    return response;
}
```

Izsek kode 4.14: Izsek implementacije za klic spletne storitve *ProcessForm*.

4.7 Primer uporabe prototipa

Če nadaljujemo s primerom uporabe iz poglavja 3.7.7, kjer smo opisali vhodne podatke, na podlagi katerih želimo izvesti integracijo, v tem delu pričakujemo njen rezultat, torej spletni uporabniški vmesnik.

Pred izvedbo integracijo je treba postoriti še nekaj stvari, datoteko z opisom uporabniškega vmesnika je treba vključiti v model. To storimo v programskem modulu *middleware*, ki ima nalogo branja vhodnih podatkov, torej njene vsebine. Poskrbeti je treba, da bodo vse spletne storitve, ki so opisane znotraj datoteke, dostopne in na voljo za uporabo. Ko imamo izpolnjene omenjene pogoje, lahko začnemo izvajanje integracije. Slednje naredimo implicitno, ko aplikacijo naložimo na aplikacijski strežnik. V primeru uspešnega nalaganja v brskalnik vnesemo naslov, kjer je dostopna osrednja aplikacija za izvajanje integracije, ki nam najprej ponudi spletno stran za izbiro uporabniškega vmesnika (Slika 4.4).



Slika 4.4: Izbira uporabniškega vmesnika.

Po izbiri uporabniškega vmesnika se v ozadju izvedeta priprava podatkov in postopek integracije, katerega rezultat je integriran uporabniški vmesnik. Naslednji korak sta vpis podatka in njegova oddaja s klikom na potrditveni gumb. Če je bila oddaja uspešna, če je torej spletna storitev uspešno zaključila obdelavo podatkov, se uporabniku prikaže spletna stran s potrditvenim statusom, sicer pa se uporabniku na vmesniku prikažejo opisi napak, ki jih je storil pri vnosu podatkov ali pa obvestilo o neuspešni oddaji podatkov. S tem smo zaključili prikaz uporabe prototipa, ki je zasnovan tako, da uporabnika vodi skozi postopek integracije, saj je prilagojen potrebam prototipa, torej testiranju.

5 Evalvacija pripravljene rešitve

Postopek evalvacije modela smo izvedli s pomočjo pripravljene rešitve, v katero smo vključili štiri uporabniške vmesnike aplikacij SaaS, ki predstavljajo nabor poslovnih aplikacij. S primerom smo prikazali ustreznost modela in njegove rešitve glede na začrtane izzive in cilje magistrskega dela. Z izbiro aplikacij smo pokrili izbrane scenarije, ki se pogosto pojavljajo v realnih projektih. Definicijo opisnega jezika WUIIDL smo preverili predvsem z vidika koncepta jezika in namena njegove uporabe. Tovrstni jezik je sicer težko oceniti brez objektivnih ocen, za katere je treba preveriti večino od možnih scenarijev, ki se lahko pojavijo pri integraciji spletnih uporabniških vmesnikov.

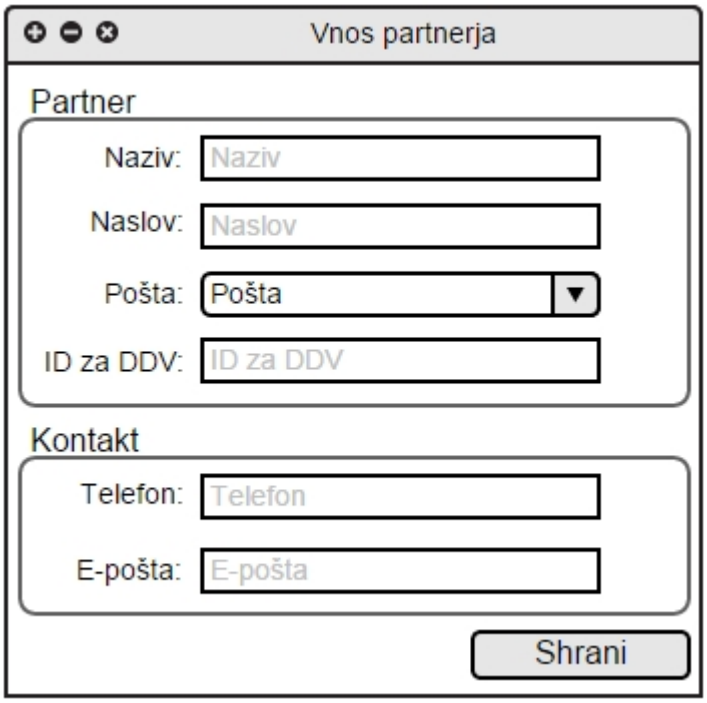
Najprej bomo podrobneje opisali primer uporabe in predstavili scenarije, za katere smo pripravili opise uporabniških vmesnikov. Pri evalvaciji modela smo se osredotočili predvsem na opise komponent in pripadajoče podatke. Podatkom smo namreč namenili posebno pozornost na področju njihovih vrednosti, saj smo v opisih opredelili posebne omejitve, ki bi lahko v nadaljevanju predstavljale potencialno oviro. Preverili smo tudi ustreznost modela s stališča komunikacije med posameznimi komponentami modela, ki smo jo izvedli s pomočjo spletnih storitev. S tem smo ocenili ustreznost koncepta, ki za potrebe posameznega projekta dopušča tudi možnost nadgradnje.

5.1 Primer uporabe

Za evalvacijo modela smo podprli nekatere najbolj pogoste poslovne procese v podjetju, ki se ukvarja s predelavo živil, v našem primeru s predelavo sadja. Podprli smo proces za dodajanje partnerjev, s katerimi podjetje posluje in so ključnega pomena za njegovo uspešnost. Glede na to, da smo za primer vzeli podjetje, ki se ukvarja s predelavo, smo udejanjili proces dodajanja dobaviteljev, ki so hkrati tudi poslovni partnerji in skrbijo za nemoteno oskrbo s surovinami. Podprli smo tudi poslovni proces naročanja surovin in njihovega transporta. S tem smo pokrili nekaj osnovnih procesov, ki so običajno prisotni v industrijskih panogah.

5.1.1 Poslovni proces za vnos partnerja

Opis uporabniškega vmesnika (Slika 5.1) za vnos poslovnih partnerjev zajema osnovne podatke o partnerju, s pomočjo katerih ga uporabnik lahko kasneje glede na izbrane kriterije poišče v bazi. Podatki so razdeljeni v dve skupini podatkov: prva zajema splošne podatke o partnerju (*Partner*), druga pa vsebuje informacije o kontaktih (*Kontakt*).



Slika 5.1: Skica vmesnika za dodajanje partnerja.

Na podlagi tega smo opis komponent za vnos podatkov prav tako razdelili na dve skupini podatkov, in sicer skupino z nazivom *Partner* (Izsek kode 5.1) in skupino z nazivom *Kontakt*.

```
<ft:fieldGroup>
      <cmn:groupName>Partner</cmn:groupName>
      <cmn:fieldList>
            <cmn:field>
                  <cmn:fieldName>Naziv</cmn:fieldName>
                  <cmn:fieldType>Text</cmn:fieldType>
                  <cmn:fieldValidationList>
                        <cmn:validator>
                              <cmn:type>NotNull</cmn:type>
                        </cmn:validator>
                        <cmn:validator>
                              <cmn:type>NotEmpty</cmn:type>
                        </cmn:validator>
                  </cmn:fieldValidationList>
            </cmn:field>
            <!-- ... -->
      </cmn:fieldList>
</ft:fieldGroup>
```

Izsek kode 5.1: Izsek definicije, opis skupine *Partner*.

Med osnovne podatke partnerja smo kot prvega dodali naziv podjetja (*Naziv*) (Izsek kode 5.1). Obliko za vnos podatka smo določili kot klasično vnosno polje za vnos znakovnih nizov. S pomočjo opisa pravil za validacijo smo definirali tudi, da je podatek obvezen in ne sme biti krajši od enega znaka. Eden izmed osnovnih podatkov za vodenje poslovnih partnerjev je tudi

naslov podjetja (*Naslov*). Opisali smo ga na podoben način kot naziv partnerja, razlikujeta se le v poimenovanju. Pri opisu komponente za vnos informacije o pošti (*Pošta*) (Izsek kode 5.2) pa smo spremenili obliko vnosa, in sicer smo izbrali spustni seznam, za katerega smo pred tem določili možnosti, med katerimi uporabnik lahko izbira. Vpisali smo namreč nekaj poštnih številk skupaj z nazivi, tako da lahko uporabnik enostavno izbere eno izmed njih. Vnos informacije o pošti je obvezen, vrednost pa ne sme biti krajša od enega znaka.

```
<cmn:field>
      <cmn:fieldName>Pošta</cmn:fieldName>
      <cmn:fieldType>DropDown</cmn:fieldType>
      <cmn:fieldValidationList>
            <cmn:validator>
                  <cmn:type>NotNull</cmn:type>
            </cmn:validator>
            <cmn:validator>
                  <cmn:type>NotEmpty</cmn:type>
            </cmn:validator>
      </cmn:fieldValidationList>
      <cmn:fieldDataSource>
            <cmn:WSendpoint>
                  <cmn:address>
                        http://localhost:8282/dataWS/GetData
                  </cmn:address>
                  <cmn:serviceName>GetData</cmn:serviceName>
                  <cmn:portType>GetData</cmn:portType>
                  <cmn:operation>getPostNumbers</cmn:operation>
            </cmn:WSendpoint>
      </cmn:fieldDataSource>
</cmn:field>
```

Izsek kode 5.2: Izsek definicije, opis vnosnega polja Pošta.

Nazadnje smo dodali še opis komponente za vnos davčne številke (*ID za DDV*). Podatek bo moral uporabnik vpisati v obliki znakovnega niza in je prav tako obvezen, le da smo tokrat določili tudi omejitev dolžine vnosa, in sicer na dolžino natančno petih znakov. Tako lahko opazimo, da smo vse podatke označili kot obvezne, zato bo moral uporabnik za uspešno oddajo podatkov izpolniti vsa vnosna polja.

V skupini kontaktnih podatkov (*Kontakt*) (Izsek kode 5.3) smo opisali najbolj običajne tipe kontaktnih podatkov, kot sta telefonska številka (*Telefon*) in e-poštni naslov (*E-pošta*). Pri vnosu telefonske številke smo želeli preveriti tudi dolžino telefonske številke, zato smo dolžino omejili na najmanj deset znakov. Na takšen način smo posredno določili tudi obveznost podatka, saj bo katerakoli vrednost, krajša od desetih znakov, pri izvajanju validacije zavrnjena. Oba podatka bo uporabnik moral vpisati v polje za vnos znakovnih nizov, pri e-poštnem naslovu pa smo nastavili tudi preverjanje formata vrednosti, ki je za e-poštni naslov posebej predpisan. S širokim izborom pravil za izvajanje validacije smo preverili predvsem njihovo delovanje in obnašanje v različnih primerih.

```
<ft:fieldGroup>
      <cmn:groupName>Kontakt</cmn:groupName>
      <cmn:fieldList>
            <cmn:field>
                  <cmn:fieldName>Telefon</cmn:fieldName>
                  <cmn:fieldType>Text</cmn:fieldType>
                  <cmn:fieldValidationList>
                        <cmn:validator>
                              <cmn:type>NotNull</cmn:type>
                        </cmn:validator>
                        <cmn:validator>
                              <cmn:type>Length</cmn:type>
                              <cmn:value>min=10</cmn:value>
                        </cmn:validator>
                  </cmn:fieldValidationList>
            </cmn:field>
            <cmn:field>
                  <cmn:fieldName>E-pošta</cmn:fieldName>
                  <cmn:fieldType>Text</cmn:fieldType>
                  <cmn:fieldValidationList>
                        <cmn:validator>
                              <cmn:type>NotNull</cmn:type>
                        </cmn:validator>
                        <cmn:validator>
                              <cmn:type>Pattern</cmn:type>
                              <cmn:value>^[_A-Za-z0-9-]+(.[_A-Za-z0-9-]+)*@
                                    [A-Za-z0-9-]+(.[A-Za-z0-9-]+)*(.[A-Za-z]
                                    {2,})$
                              </cmn:value>
                        </cmn:validator>
                  </cmn:fieldValidationList>
            </cmn:field>
      </cmn:fieldList>
</ft:fieldGroup>
```

Izsek kode 5.3: Izsek definicije, opis skupine *Kontakt*.

5.1.2 Poslovni proces za dodajanje dobaviteljev

Pri opisu uporabniškega vmesnika za dodajanje dobaviteljev smo bili posebej pozorni na dve skupini podatkov, in sicer skupino z osnovnimi podatki o dobavitelju, ki v tem primeru ponovno predstavlja informacije o partnerju (*Partner*). Druga skupina podatkov pa vključuje predvsem informacije za potrebe finančne službe (*Poslovanje*). Med osnovnimi podatki dobaviteljev so sicer nekateri podatki, ki se glede na prejšnji opis poslovnega procesa za dodajanje partnerjev ponovijo, vendar le s to razliko, da smo v primeru dobavitelja želeli pridobiti tudi informacijo o kontaktni osebi (*Kontaktna oseba*). Zanjo smo določili obliko za vnos znakovnega niza in zahtevali obvezno izpolnjevanje ter omejitev dolžine vrednosti na najmanj en znak.

Za podjetje, ki bo izvajalo dobavo surovin, smo želeli izvedeti tudi obliko podjetja (*Oblika podjetja*), s katero podjetje nastopa na trgu. V ta namen smo določili polje (Izsek kode 5.4) z izbirnimi gumbi in vnaprej ponujenimi možnostmi, med katerimi lahko uporabnik izbira.

```
<cmn:field>
      <cmn:fieldName>Oblika podjetja</cmn:fieldName>
      <cmn:fieldType>Radio</cmn:fieldType>
      <cmn:fieldValidationList>
            <cmn:validator>
                  <cmn:type>NotNull</cmn:type>
            </cmn:validator>
            <cmn:validator>
                  <cmn:type>NotEmpty</cmn:type>
            </cmn:validator>
      </cmn:fieldValidationList>
      <cmn:fieldDataSource>
            <cmn:WSendpoint>
                  <cmn:address>
                        http://localhost:8282/dataWS/GetData
                  </cmn:address>
                  <cmn:serviceName>GetData</cmn:serviceName>
                  <cmn:portType>GetData</cmn:portType>
                  <cmn:operation>getCompanyForms </cmn:operation>
            </cmn:WSendpoint>
      </cmn:fieldDataSource>
</cmn:field>
```

Izsek kode 5.4: Izsek definicije, opis vnosnega polja *Oblika podjetja*.

Skupina podatkov za finančno službo obsega dva najbolj pogosta tovrstna podatka, in sicer podatek o transakcijskem računu (*Račun*) in rok plačila, izražen v številu dni (*Rok plačila*). Za vnos transakcijskega računa smo izbrali komponento za vnos znakovnega niza in omogočili dodatno preverjanje formata vrednosti, saj so tega predpisale bančne institucije. V primeru opisa komponente za izpolnjevanje števila dni za poravnavo obveznosti smo izbrali polje za vnos številske vrednosti.

```
<cmn:validator>
    <cmn:type>Range</cmn:type>
    <cmn:value>min=1;max=31</cmn:value>
</cmn:validator>
```

Izsek kode 5.5: Izsek definicije, omejitev vrednosti podatka.

Poleg tega smo zahtevali obvezen vnos ter omejitev vrednosti (Izsek kode 5.5), in sicer tako navzdol kot tudi navzgor. V našem primeru smo dovolili vrednosti le med 1 in 31, torej razliko enega meseca. S tem smo zaključili opis vnosnih polj za zbiranje informacij o dobaviteljih, ki smo jih razporedili v dve ločeni skupini podatkov.

5.1.3 Poslovni proces za oddajo naročil

Za opis uporabniškega vmesnika, ki bo ponujal podporo poslovnemu procesu za oddajo naročil surovin, smo vključil dve skupini podatkov. Prva skupina vsebuje podatke iz naslova partnerja oz. dobavitelja (*Partner*), druga skupina pa je namenjena opisu naročila (*Naročilo*) (Izsek kode 5.6). Skupina, ki vsebuje podatke o dobavitelju, je povsem enaka skupini, ki smo jo predstavili v procesu dodajanja dobavitelja.

```
<ft:fieldGroup>
      <cmn:groupName>Naročilo</cmn:groupName>
      <cmn:fieldList>
            <cmn:field>
                  <cmn:fieldName>Artikel</cmn:fieldName>
                  <cmn:fieldType>DropDown</cmn:fieldType>
                  <cmn:fieldValidationList>
                        <cmn:validator>
                              <cmn:type>NotNull</cmn:type>
                        </cmn:validator>
                        <cmn:validator>
                              <cmn:type>NotEmpty</cmn:type>
                        </cmn:validator>
                  </cmn:fieldValidationList>
                  <cmn:fieldDataSource>
                        <cmn:WSendpoint>
                              <cmn:address>
                                    http://localhost:8282/dataWS/GetData
                              </cmn:address>
                              <cmn:serviceName>GetData</cmn:serviceName>
                              <cmn:portType>GetData</cmn:portType>
                              <cmn:operation>getProductList</cmn:operation>
                        </cmn:WSendpoint>
                  </cmn:fieldDataSource>
            </cmn:field>
            <cmn:field>
                  <cmn:fieldName>Količina</cmn:fieldName>
                  <cmn:fieldType>Number</cmn:fieldType>
                  <cmn:fieldValidationList>
                        <cmn:validator>
                              <cmn:type>NotNull</cmn:type>
                        </cmn:validator>
                        <cmn:validator>
                              <cmn:type>Range</cmn:type>
                              <cmn:value>min=1;max=20</cmn:value>
                        </cmn:validator>
                  </cmn:fieldValidationList>
            </cmn:field>
      </cmn:fieldList>
</ft:fieldGroup>
```

Izsek kode 5.6: Izsek definicije, opis skupine *Naročilo*.

Skupina podatkov o naročilu vključuje informacijo o naročeni surovini (*Artikel*). Polje smo predstavili s pomočjo komponente s spustnim seznamom, ki smo ji vnaprej podali možnosti,

med katerimi lahko izbira uporabnik. Poleg tega je podatek obvezen, zaradi varnosti pa smo določili tudi omejitev, in sicer dolžina niza ne sme biti krajša od enega znaka. Ob informaciji o naročeni surovini pa smo potrebovali tudi podatek o količini naročila, to je koliko surovine želi uporabnik naročiti. V ta namen smo dodali opis komponente za vnos informacije o količini naročila (*Količina*) izbrane surovine. Kot obliko vnosa smo določili komponento, ki dovoljuje le vnos številskih vrednosti in je temu prilagojena, saj ima dodane gumbe za višanje oz. nižanje vrednosti z določenim korakom. V našem primeru smo izbrali korak vrednosti 1, torej bo uporabnik s klikom na gumb spremenil trenutno vrednost za 1 navzgor oz. navzdol, odvisno od izbranega gumba. Glede na to, da smo za omenjeni podatek predvideli številko, tj. vrednost, smo to želeli tudi omejiti, podobno kot v običajnem poslovnem procesu. Količino naročene surovine smo omejili na najmanj eno količinsko enoto in največ 20 količinskih enot (Izsek kode 5.6). V podjetju se omejitev določa na podlagi predhodnih analiz. Za podporo poslovnega procesa oddaje naročila smo torej pripravili vmesnik z informacijami o dobavitelju in naročilu.

5.1.4 Poslovni proces za naročila transporta

Podobno kot za ostale poslovne procese smo tudi za naročilo transporta pripravili svoj uporabniški vmesnik. Pri transportu nas običajno zanimata dve pomembnejši informaciji, in sicer podjetje, ki bo opravilo transport surovine, in v kolikšnem času bo surovina dostavljena na cilj. Podjetje, ki bo opravilo transport, lahko označimo kot partnerja naročnika transporta. Zato smo komponente uporabniškega vmesnika za potrebe naročanja transporta razdelili v dve skupini, in sicer skupino z informacijami o izvajalcu transporta (*Partner*) in skupino z informacijami o transportu (*Transport*). Tako kot že pri predhodnem uporabniškem vmesniku smo tudi v tem primeru skupino komponent s podatki o izvajalcu transporta definirali zelo podobno skupini komponent v preostalih vmesnikih.

Skupina komponent s podatki o transportu vključuje informacije o načinu prevoza (*Način prevoza*) (Izsek kode 5.7), zato smo komponento za izbiro načina transporta predstavili s spustnim seznamom, s katerega bo uporabnik lahko izbral eno izmed opcij. Za omenjeno komponento smo tako pripravili nabor možnosti, ki bodo na voljo uporabniku. Podatek je obvezen in zaradi varnosti omejen z dolžino najmanj enega znaka.

```
<cmn:field>
      <cmn:fieldName>Način prevoza</cmn:fieldName>
      <cmn:fieldType>DropDown</cmn:fieldType>
      <cmn:fieldValidationList>
            <cmn:validator>
                  <cmn:type>NotNull</cmn:type>
            </cmn:validator>
            <cmn:validator>
                  <cmn:type>NotEmpty</cmn:type>
            </cmn:validator>
      </cmn:fieldValidationList>
      <cmn:fieldDataSource>
            <cmn:WSendpoint>
                  <cmn:address>
                        http://localhost:8282/dataWS/GetData
                  </cmn:address>
                  <cmn:serviceName>GetData</cmn:serviceName>
                  <cmn:portType>GetData</cmn:portType>
                  <cmn:operation>getTransportTypeList</cmn:operation>
            </cmn:WSendpoint>
      </cmn:fieldDataSource>
</cmn:field>
```

Izsek kode 5.7: Izsek definicije, opis vnosnega polja *Način prevoza*.

Poleg načina prevoza pa naročnika transporta zanima še podatek o času (*Trajanje*), ki je potreben za dostavo surovine. V ta namen smo skupini komponent pridružili opis komponente (Izsek kode 5.8), v katero bo naročnik lahko vpisal časovno omejitev dobave, ki je zanj še sprejemljiva. Uporabniku smo omogočili vnos vrednosti s pomočjo polja za vnos številskih vrednosti, ki vsebuje dodatna gumba za višanje in nižanje vrednosti. Podatek bo uporabnik moral obvezno izpolniti, poleg tega je omejen na najmanjšo vrednost 1. S tem uporabniku preprečimo vnos negativnih števil in ničelne vrednosti, ki v tem primeru nimajo pravega pomena.

```
<cmn:field>
    <cmn:fieldName>Trajanje</cmn:fieldName>
    <cmn:fieldType>Number</cmn:fieldType>
    <cmn:fieldValidationList>
        <cmn:validator>
            <cmn:type>NotNull</cmn:type>
        </cmn:validator>
        <cmn:validator>
            <cmn:type>Range</cmn:type>
            <cmn:value>min=1</cmn:value>
        </cmn:validator>
    </cmn:fieldValidationList>
</cmn:field>
```

Izsek kode 5.8: Izsek definicije, opis vnosnega polja *Trajanje*.

Ponudnike transportnih storitev običajno zanima tudi informacija o obliki tovora (*Oblika tovora*) (Izsek kode 5.9), na podlagi katere lahko prevozno sredstvo čim bolj prilagodijo tovoru.

S tem bo naročnik transporta izvajalcu podal informacijo o obliki tovora in njegove posebnosti, na katere bo moral biti med prevozom še posebno pozoren. Lastnosti tovora se med seboj ne izključujejo, zato smo za vnosno polje izbrali komponento za izbiro možnosti s pomočjo potrjevalnih polj. Uporabnik bo torej lahko izbral več lastnosti hkrati, in sicer v katerikoli kombinaciji.

```
<cmn:field>
      <cmn:fieldName>Oblika tovora</cmn:fieldName>
      <cmn:fieldType>CheckBox</cmn:fieldType>
      <cmn:fieldValidationList>
            <cmn:validator>
                  <cmn:type>NotNull</cmn:type>
            </cmn:validator>
      </cmn:fieldValidationList>
      <cmn:fieldDataSource>
            <cmn:WSendpoint>
                  <cmn:address>
                        http://localhost:8282/dataWS/GetData
                  </cmn:address>
                  <cmn:serviceName>GetData</cmn:serviceName>
                  <cmn:portType>GetData</cmn:portType>
                  <cmn:operation>getCargoTypeList</cmn:operation>
            </cmn:WSendpoint>
      </cmn:fieldDataSource>
</cmn:field>
```

Izsek kode 5.9: Izsek definicije, opis vnosnega polja *Oblika tovora*.

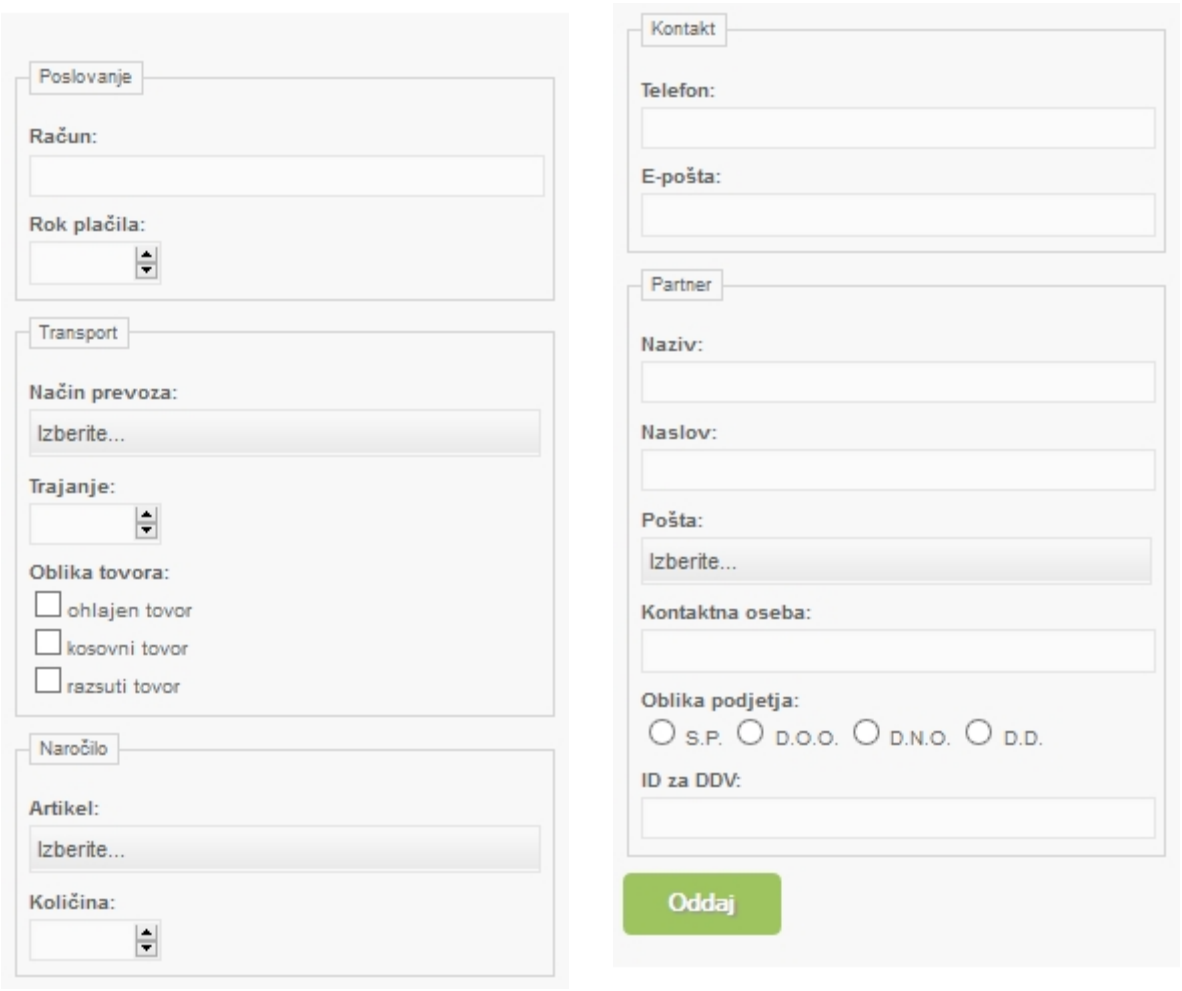
5.2 Postopek integracije

Model za integracijo spletnih uporabniških vmesnikov smo preverili najprej v smislu podpore vseh zahtev, ki smo jih podali ob začetku razvoja modela. Tako smo definirali naslednje cilje modela:

- definicija jezika za opis uporabniških vmesnikov in njihovih stičnih točk;
- integracija spletnih uporabniških vmesnikov na podlagi podanih opisov;
- podpora statične validacije vnesenih podatkov;
- komuniciranje z zunanjimi aplikacijami s pomočjo spletnih storitev.

Za začetek smo preverili definicijo opisnega jezika WUIIDL, s katerim smo pripravili opise posameznih uporabniških vmesnikov. Pri tem smo ugotovili, da jezik omogoča dovolj široko paleto možnosti za opis uporabniških vmesnikov in pripadajočih dogodkov. To smo ugotovili s pregledom opisanih vmesnikov, ki so bili pripravljeni za podporo izbranim scenarijem. Koncept jezika je tako zasnovan dovolj široko, da je z njim možno opisati večino uporabniških vmesnikov, namenjenih različnim scenarijem.

Postopek za integracijo spletnih uporabniških vmesnikov, ki vključuje generiranje vmesnikov (Slika 5.2) na podlagi podanih opisov, smo preverili s pomočjo tabele, v katero smo na eni strani vnesli ključne podatke vmesnika, na drugi strani pa scenarije oz. vmesnike.



Slika 5.2: Primer integriranega uporabniškega vmesnika.

Na koncu smo rezultate integracije primerjali s pričakovanimi rezultati v tabeli (Tabela 5.1). Na takšen način smo lahko ugotovili, ali je v primeru izbrane kombinacije parametrov komponenta ustrezno pripravljena. S tem ugotavljamo, da algoritem za izvedbo integracije deluje pravilno in upošteva implementirana pravila, na podlagi katerih smo definirali postopek integracije.

	Dodajanje partnerja	***Dodajanje dobavitelja***	***Oddaja naročila***	***Oddaja transporta***
Partner				
Naziv	✓	✓	✓	✓
Naslov	✓	✓	✓	✓
Pošta	✓	✓	✓	✓
ID za DDV	✓			
Kontaktna oseba		✓	✓	
Oblika podjetja		✓	✓	
Kontakt				
Telefon	✓			
E-pošta	✓			
Poslovanje				
Račun		✓		
Rok plačila		✓		
Transport				
Način prevoza				✓
Trajanje				✓
Oblika tovora				✓
Naročilo				
Artikel			✓	
Količina			✓	

Tabela 5.1: Pregled polj za integracijo.

Iz tabele je razvidno, v katerih primerih se mora pojaviti komponenta za vnos določenega podatka. Pogoj je, da mora vsaj pri enem izmed izbranih scenarijev oz. vmesnikov obstajati opis za opazovano komponento. V tabeli je nadalje možno ugotoviti tudi skupino, ki ji pripada izbrani podatek. Preverimo lahko tudi uspešnost izvedene integracije skupin v primeru izbrane kombinacije scenarijev. Ocene smo lahko podali šele po pregledu rezultata, torej integriranega uporabniškega vmesnika. Na njem smo lahko opazili skupine komponent, ki so predstavljene v izbranih opisih vmesnikov.

Omenili smo že, da smo preverili model tudi s stališča validacije, kjer poteka pomembno preverjanje vnesenih podatkov, saj bi v primeru neustreznih podatkov prišlo do težav pri oddaji podatkov s pomočjo opisanih spletnih storitev. V našem prototipu nastopa le statična validacija (Slika 5.3), ki pa smo jo vseeno razdelili na dva dela, in sicer validacijo znakovnih nizov ter validacijo numeričnih vrednosti. V primeru validacije znakovnih nizov smo preverili obnašanje prototipa s pomočjo različnih pravil. Njihov nabor smo zbrali v tabeli (Tabela 5.2), dodali smo še dejanske primere vrednosti, ki so pričakovani s strani vmesnika, in tako zgradili pravilnostno tabelo, ki nam sporoča informacije o dopustnih vrednostih pri izbranih pravilih validacije. Na podlagi omenjene tabele smo izvedli tudi verifikacijo delovanja izbranega nabora pravil validacije. S tem smo lahko zagotovili ustreznost implementacije postopkov za izvajanje validacije in tako potrdili koncept modela, ki opisuje njeno delovanje.

Slika 5.3: Primer rezultatov validacije.

Iz tabele (Tabela 5.2) lahko nadalje ugotovimo, da nekaterih pravil validacije ni možno aplicirati na določene tipe vrednosti. Tako na primer numeričnih vrednosti ni možno preverjati s pravilom za vzorčno preverjanje (*Pattern*), tako kot tudi ni možno ugotavljati njihove dolžine zapisa, saj je v ta namen na voljo pravilo za preverjanje dovoljenega območja (*Range*), kjer lahko določimo natančne vrednosti za zgornjo in spodnjo omejitev. Prav zaradi teh raznolikosti smo opisom vmesnikov dodali različne kombinacije in tako ugotovili pravilnost delovanja validacije podatkov.

		null	*»«*	*» «*	*»test«*	*»SI5602001234567«*	*»test@test.si«*	*7*	*25*
NotNull		✘	✓	✓	✓	✓	✓	✓	✓
NotEmpty		✘	✘	✓	✓	✓	✓	-	-
Pattern									
value	email	✘	✘	✘	✘	✘	✓	-	-
value	IBAN	✘	✘	✘	✘	✓	✘	-	-
Length									
min	1	✘	✘	✓	✓	✘	✘	-	-
max	5								
Range									
min	1	-	-	-	-	-	-	✓	✘
max	20								

Tabela 5.2: Tabela s pravili validacije.

Na koncu smo preverili tudi način in pravilnost delovanja komunikacije s spletnimi storitvami, ki predstavljajo ključen del integracije ob oddaji podatkov. Pri tem smo posebej preverjali ločevanje podatkov, ki jih je oddal integrirani vmesnik, v izvorne skupine, ki so določene z opisi vmesnikov oz. aplikacij. V ta namen smo pripravili posebno spletno storitev, ki smo ji za vsak poslovni proces definirali svojo operacijo, ki smo jo nato dodali v opis spletne storitve (Izsek kode 5.10).

```
<ft:WSendpoint>
      <cmn:address>
            http://localhost:8282/testWS/ProcessForm
      </cmn:address>
      <cmn:serviceName>ProcessForm</cmn:serviceName>
      <cmn:portType>ProcessForm</cmn:portType>
      <cmn:operation>processTransport</cmn:operation>
</ft:WSendpoint>
```

Izsek kode 5.10: Izsek definicije, opis spletne storitve.

Po uspešno opravljenem ločevanju podatkov v skupine smo nato vsako izmed njih poskusili oddati na določeno spletno storitev (Slika 5.4), in sicer pripadajoči operaciji. S pomočjo posebne beležke spletne storitve, ki beleži vsebino vsakega zahtevka, smo na koncu preverili zapise poslanih skupin podatkov skupaj z njihovimi vrednostmi. Tako smo lahko ugotovili, ali so podatki pravilno združeni v skupine in ali ustrezajo vsem omejitvam, ki smo jih podali s posameznimi opisi vmesnikov.

Slika 5.4: Prikaz rezultata oddaje podatkov.

S tem smo sklenili krog, ki obsega integracijo spletnih uporabniških vmesnikov, preverjanje in oddajo podatkov. Na tem mestu ugotavljamo, da koncept, na podlagi katerega smo zasnovali model in zanj pripravili prototip, ustreza zastavljenim ciljem. Z njim smo preverili ustreznost in pripravljenost na izvedbo določenih pristopov.

6 Sklepne ugotovitve

V magistrskem delu smo definirali model za integracijo spletnih uporabniških vmesnikov. Najprej smo pregledali področje uporabniških vmesnikov, kakšne so njihove lastnosti, kakšni pristopi so trenutno uveljavljeni in katere vrste integracij že poznamo. Pri tem smo se posvečali podobnostim in razlikam uporabniških vmesnikov, ob tem pa razmišljali o svojem cilju, to je definiciji koncepta, s katerim bi lahko na relativno enostaven način razvijalcem zvišali stopnjo učinkovitosti razvoja in jih pri tem dodatno spodbudili k uporabi že obstoječih konceptov. Pregledali smo tudi obstoječe tehnologije in rešitve, ki so najbolj sorodne našemu izzivu. Pregled stanja smo zaključili z naborom izzivov, s katerimi smo se kasneje srečali.

Definirali smo koncept modela, pri katerem je glavnega pomena opisni jezik, s katerim je možno natančno opisati ključne komponente uporabniškega vmesnika in njegove predvidene dogodke. Jezik smo razdelili na več sklopov, in sicer smo definirali sklop, namenjen opisu komponent, prek katerih poteka interakcija uporabnika z uporabniškim vmesnikom. Nato smo definirali sklop, v katerem je moč opisati predvidene postopke validacije, na koncu pa smo določili še sklop za komunikacijo modela z zunanjim svetom. Komuniciranje smo opredelili s pomočjo spletnih storitev, ki skrbijo za obdelavo dostavljenih informacij. Poskrbeli smo tudi za definicijo načina prikaza opisanih komponent uporabniškega vmesnika oz. dejansko izvedbo integracije, ki skrbi za pravilno interpretacijo pravil, nastalih ob razvoju modela. Zaradi omejenega obsega magistrskega dela in tudi časa, ki smo ga imeli na voljo, smo venomer razmišljali tudi o izboljšavah. Zato smo na nekaterih mestih opisali tudi pomanjkljivosti trenutnega modela in predlagane izboljšave oz. razširitve.

V naslednjem koraku smo opisali implementacijo rešitve, ki smo jo pripravili po opisanem modelu. Najprej smo pozornost usmerili v razvoj opisnega jezika, ki je med drugim eden pomembnejših prispevkov magistrskega dela. Zanj smo izbrali primerne tehnologije, kot na primer sheme XSD, s katerimi smo predpisali obliko strukture podatkov. Tako smo definirali opisni jezik, s katerim lahko opišemo ključne podrobnosti uporabniških vmesnikov, ki jih želimo integrirati. Nato smo izbrali primerne tehnologije, s pomočjo katerih smo lahko implementirali začrtani koncept, pri tem pa smo bili posebej pozorni na njihovo medsebojno združljivost. Izbrali smo programski jezik Java, v katerem smo pripravili prototip. Za potrebe razvojnega okolja smo izbrali programsko opremo Eclipse, s pomočjo katere smo pripravili razvojno okolje za pomoč pri razvoju programske rešitve. Glede na to, da smo pripravljali dokaj obširno rešitev, smo uporabili tudi programsko ogrodje Apache Maven, za lažje upravljanje projekta. Prednost uporabe omenjenega ogrodja je tudi v boljšem pregledu in nadzoru projekta. Pred začetkom implementacije prototipa smo za izvedbo integracije spletnega uporabniškega vmesnika izbrali še primerno ogrodje, s pomočjo katerega smo kasneje lažje implementirali prikaz komponent na vmesniku, vključno z njihovimi odzivi in dogodki. V ta namen smo izbrali ogrodje Apache Wicket, pri katerem smo preizkusili njegove prednosti, hkrati pa se spoprijeli

tudi z njegovimi slabostmi. Nekajkrat smo namreč naleteli na težave pri implementaciji dinamičnega prikazovanja komponent, saj Apache Wicket, tako kot druga ogrodja, v ta namen uporablja specifične nastavke, s katerimi se je za napredno uporabo treba najprej spoznati.

Za uspešen zaključek prototipa smo pozornost namenili tudi implementaciji postopka za preverjanje vnesenih podatkov, in sicer glede na zahtevane omejitve, ki smo jih predvideli v strukturi uporabniškega vmesnika. Za tem je prišel na vrsto ključni del, in sicer izvedba oddaje preverjenih podatkov na končen cilj, ki ga v našem primeru zastopa delujoča spletna storitev. Brez uspešne oddaje podatkov, ki jih je uporabnik vnesel v integriran uporabniški vmesnik, postopek integracije izgubi svoj pomen. Po oddaji podatkov smo poskrbeli še za obvestilo uporabniku o rezultatu omenjene faze.

Ob zaključku magistrskega dela smo preverili prispevke, tako teoretične kot tudi praktične. Za potrebe preverjanja smo pripravili opise uporabniških vmesnikov na podlagi vnaprej izbranih scenarijev za izbran primer uporabe. Pri izboru scenarijev smo poskušali doseči čim boljšo pokritost teh, saj smo kasneje opravili analizo prav na podlagi omenjene pokritosti in tako pridobili objektivne ocene. Z njihovo pomočjo smo lahko potrdili pravilnost delovanja modela.

Implementacija rešitve je zasnovana modularno in generično, kar omogoča enostavne dopolnitve in nadgradnje. Opisni jezik lahko kdorkoli po potrebi razmeroma hitro in enostavno prilagodi, tako kot tudi celoten model. Vsakega izmed modulov pa je moč uporabiti za specifične potrebe svojega projekta. Največji prispevek magistrskega dela je definicija opisnega jezika, ki ima vlogo formalizacije vhodnih podatkov v postopek integracije.

Literatura

[1] S. Aljawarneh, F. Alkhateeb, E. Al Maghayreh, "A Semantic Data Validation Service for Web Applications", *Journal of Theoretical and Applied Electronic Commerce Research*, št. 1, zv. 5, str. 39–55, 2010.

[2] (2014) API-fication. Dostopno na: http://www.hcltech.com/sites/default/files/apis_for_dsi.pdf (pridobljeno 25. 3. 2015).

[3] (2011) Applications Integration. Dostopno na: https://www.mulesoft.com/resources/esb/applications-integration (pridobljeno 25. 1. 2015).

[4] (2009) Architectural Styles and the Design of Network-based Software Architectures. Representational State Transfer (REST). Dostopno na: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (pridobljeno 10. 2. 2015).

[5] D. Bianchini, V. De Antonellis, M. Melchiori, "Towards Semantic-assisted Web Mashup generation", v zborniku *23rd International Workshop on Database and Expert Systems Applications (DEXA)*, Vienna, Italy, september 2012, str. 279–283.

[6] E. Castillejoa, A. Almeidaa, D. López-de-Ipiñaa, "Ontology-Based Model for Supporting Dynamic and Adaptive User Interfaces ", *International Journal of Human-Computer Interaction*, št. 10, zv. 30, str. 771–786, 2014.

[7] B. chul Kwon, W. Javed, N. Elmqvist, J. Soo Yi, "Direct Manipulation Through Surrogate Objects", v zborniku *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, Vancouver, Canada, maj 2011, str. 627–636.

[8] (2014) Command line vs. GUI. Dostopno na: http://www.computerhope.com/issues/ch000619.htm (pridobljeno 15. 1. 2015).

[9] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, L. Yan, "Distributed orchestration of user interfaces", *Information Systems*, št. 6, zv. 37, str. 539–556, 2011.

[10] (2009) Das SoKNOS Projekt – Ein Überblick. Dostopno na: http://www.bmbf.de/pubRD/SoKNOS_Ziegert_Auftakt_IPF_SuRvM.pdf (pridobljeno: 20. 1. 2015).

[11] (2012) Data integration. Dostopno na: http://en.wikipedia.org/wiki/Data_integration (pridobljeno 22. 4. 2015).

[12] W. Herry Utomo, T. Wellem, "The Implementation of Enterprise Service Bus (ESB) in Graduation Business Process Integration", *Journal of Theoretical and Applied Information Technology*, št. 2, zv. 62, str. 364–370, 2014.

[13] E. Horvitz, C. M. Kadie, T. Paek, D. Hovel, "Models of Attention in Computing and Communication: From Principles to Applications", *Communications of the ACM*, št. 3, zv. 46, str. 52–59, 2003.

[14] R. Kennard, J. Leaney, "Is there convergence in the field of UI generation?", *Journal of Systems and Software,* št. 12, zv. 84*,* str. 2079–2087, 2011.

[15] R. Kennard, J. Leaney, "Towards a general purpose architecture for UI generation", *Journal of Systems and Software*, št. 10, zv. 83*,* str. 1896–1906, 2010.

[16] T. Lei, W. Yang, T. Wang, "Design and Optimization of Application Integration Based on Notification Service", v zborniku *3rd International Conference on Software Engineering and Service Science (ICSESS)*, Beijing, China, junij 2012, str. 224–227.

[17] S. Lepreux, J. Vanderdonckt, C. Kolski, "User Interfaces Composition with UsiXML", v zborniku *1st International Workshop on User Interface eXtensible Markup Language (UsiXML)*, Berlin, Germany, junij 2010, str. 141–151.

[18] P. Long, Z. Chunyuan, L. Shenling, "The Research and Implementation of Portal-based Integration for Digital Campus", v zborniku *International Conference on E-Business and E-Government (ICEE)*, Shanghai, China, maj 2011, str. 1–4.

[19] R. Miñón, L. Moreno, P. Martínez, J. Abascal, "An approach to the integration of accessibility requirements into a user interface development method", *Science of Computer Programming*, št. 1, zv. 86, str. 58–73, 2014.

[20] M. Mott, J. Barkeloo, "Developmental Phonics Instruction with Touch User Interface Technology: Moving Toward a Multi-Sensory Approach for Grades Pre-K-2", v zborniku *World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education (E-Learn)*, Washington, ZDA, april 2004, str. 1395.

[21] (2011) Oracle Application Development Framework. Dostopno na: http://en.wikipedia.org/wiki/Oracle_Application_Development_Framework (pridobljeno 21. 4. 2015).

[22] (2012) OWL 2 Web Ontology Language. Dostopno na: http://www.w3.org/TR/owl2-overview (pridobljeno 6. 4. 2015).

[23] F. Paternò, C. Santoro, L. D. Spano, "MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments", *ACM Transactions on Computer-Human Interaction (TOCHI)*, št. 4, zv. 16, str. 219–224, 2009.

[24] H. Paulheim, F. Probst, "Application Integration on the User Interface Level: an Ontology-Based Approach", *Data and Knowledge Engineering*, št. 11, zv. 69, str. 1103–1116, 2010.

[25] (2000) Process Integration Oriented E-business Integration. Dostopno na: http://www.informit.com/articles/article.aspx?p=19742 (pridobljeno 26. 1. 2015).

[26] R. Ramanathan, T. Korte, "Software Service Architecture to Access Weather Data Using RESTful Web Services", v zborniku *International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, Hefei, China, julij 2014, str. 1–8.

[27] (2003) Resource Description Framework (RDF). Dostopno na: http://www.w3.org/RDF (pridobljeno 30. 3. 2015).

[28] (2014) SOAP. Dostopno na: http://en.wikipedia.org/wiki/SOAP (pridobljeno 19. 4. 2015).

[29] (2012) System integration. Dostopno na: http://en.wikipedia.org/wiki/System_integration (pridobljeno 22. 4. 2015).

[30] (2004) The Art of Unix Usability. History: A Brief History of User Interfaces. Dostopno na: http://www.catb.org/~esr/writings/taouu/html/ch02.html (pridobljeno 15. 1. 2015).

[31] H. Trætteberg, "UI Design without a Task Modeling Language - Using BPMN and Diamodl for Task Modeling and Dialog Design", v zborniku *2nd Conference on Human-Centered Software Engineering (HCSE) and 7th International Workshop on Task Models and Diagrams (TAMODIA)*, Pisa, Italy, september 2008, str. 110–117.

[32] (2014) User interface. Dostopno na: http://en.wikipedia.org/wiki/User_interface (pridobljeno 21. 4. 2015).

[33] (2004) Web Services Addressing (WS-Addressing). Dostopno na: http://www.w3.org/Submission/ws-addressing (pridobljeno 22. 4. 2015).

[34] (2013) Web Services Description Language. Dostopno na: http://en.wikipedia.org/wiki/Web_Services_Description_Language (pridobljeno 22. 4. 2015).

[35] M. Wimmer, A. Schauerhuber, W. Schwinger, H. Kargl, "On the Integration of Web Modeling Languages: Preliminary Results and Future Challenges", v zborniku *7th International Conference on Web Engineering (ICWE)*, Como, Italy, julij 2007, str. 255–269.

[36] Z. Wu, Y. Li, "Research on Enterprise Application Integration Based on Web", v zborniku *International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, Jilin, China, avgust 2011, str. 2221–2224.

[37] L. Xianming, L. Weihua, L. Shixian, "Using Ontology to Support Portlet Semantic Interoperability", v zborniku *First International Workshop on Education Technology and Computer Science (ETCS)*, Wuhan, Hubei, marec 2009, zv. 2, str. 534–537.

[38] (2006) XIML: A Universal Language for User Interfaces. Dostopno na: http://www.ximl.org/documents/XimlWhitePaper.pdf (pridobljeno 20. 4. 2015).

[39] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, M. Matera, "A Framework for Rapid Integration of Presentation Components", v zborniku *16th international conference on World Wide Web*, Banff, AB, Canada, maj 2007, str. 923–932.